

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

清华社“视频大讲堂”大系
网络开发视频大讲堂

1DVD



jQuery 开发

从入门到精通
配套视频讲解327节

精通

●中小典型实例232个●综合实战案例7个●学习参考手册11部●实用网页模版83类

袁江◎编著



清华大学出版社



执着于专业 精细于品质

光盘 · 配套视频讲解 (共计327节)

- 1.2.1 下载jQuery.mp4
- 1.2.2 安装jQuery.mp4
- 1.2.3 测试jQuery.mp4
- 1.3.1 对外接口单一使用更简单
- 1.3.2 链式语法让编码更快速
- 1.3.3 模仿CSS选择器让选取元素
- 1.3.4 扩展接口让jQuery更开放
- 1.5.1 jQuery对象不等于DOM对象
- 1.5.2 jQuery对象与DOM对象沟通
- 1.5.3 jQuery的ready不等于Java
- 2.1.1 ID选择器.mp4
- 2.1.2 标签选择器.mp4
- 2.1.3 类选择器.mp4
- 2.1.4 通配选择器.mp4
- 2.1.5 组选择器.mp4
- 2.2.1 包含选择器.mp4
- 2.2.2 子选择器.mp4
- 2.2.3 相邻选择器.mp4
- 2.2.4 兄弟选择器.mp4
- 2.2.5 层级选择器综合应用.mp4
- 2.2.6 解析层级选择器实现原理
- 2.3.1 特定位置选择器.mp4
- 2.3.2 指定范围选择器.mp4
- 2.3.3 排除选择器.mp4
- 2.3.4 特殊选择器.mp4
- 2.3.5 解析简单伪类选择器实现
- 2.4.1 匹配包含文本选择器.mp4
- 2.4.2 匹配包含元素选择器.mp4
- 2.4.3 包含判断选择器.mp4
- 2.4.4 解析内容过滤器实现原理
- 2.5 与元素显示状态相关的伪类
- 2.6 匹配子元素的伪类选择器
- 2.7 与表单对象相关的伪类选择
- 2.8 与表单属性相关的伪类选择
- 2.9 属性选择器.mp4
- 2.10 jQuery选择器应用优化.mp4
- 3.1.1 类过滤.mp4
- 3.1.2 下标过滤.mp4
- 3.1.3 表达式过滤.mp4
- 3.1.4 类过滤.mp4
- 3.1.5 映射.mp4
- 3.1.6 清洗.mp4
- 3.1.7 截取.mp4
- 3.2.1 向下查找后代元素.mp4
- 3.2.2 向上查找祖先元素.mp4
- 3.2.3 向上查找兄弟元素.mp4
- 3.2.4 向下查找兄弟元素.mp4
- 3.2.5 查找兄弟元素.mp4
- 3.2.6 添加查找对象.mp4
- 3.3.1 绑定前后jQuery对象
- 3.3.2 返回前一个jQuery对象
- 4.1.1 创建元素.mp4
- 4.1.2 输入文本.mp4
- 4.1.3 设置属性.mp4
- 4.2.1 内部插入.mp4
- 4.2.2 外部插入.mp4
- 4.3.1 移出.mp4
- 4.3.2 清空.mp4
- 4.3.3 分离.mp4
- 4.4 克隆内容.mp4
- 4.5 替换内容.mp4
- 4.6.1 外包.mp4
- 4.6.2 内包.mp4
- 4.6.3 总包.mp4
- 4.6.4 卸包.mp4
- 4.7.1 设置属性.mp4
- 4.7.2 访问属性.mp4
- 4.7.3 删除属性.mp4
- 4.8.1 添加样式.mp4
- 4.8.2 删除样式.mp4
- 4.8.3 切换样式.mp4
- 4.8.4 判断样式.mp4
- 4.9.1 读写HTML.mp4
- 4.9.2 读写文本.mp4
- 4.9.3 读写值.mp4
- 4.10.1 读写CSS样式.mp4
- 4.10.2 绝对定位.mp4
- 4.10.3 相对定位.mp4
- 4.10.4 设置大小.mp4
- 4.11 访问文档树.mp4
- 5.1.1 原始事件模型.mp4
- 5.1.2 DOM事件模型.mp4
- 5.1.3 IE事件模型.mp4
- 5.2.1 Event对象.mp4
- 5.2.2 事件流.mp4
- 5.2.3 事件控制.mp4
- 5.3.1 注册事件.mp4
- 5.3.2 注销事件.mp4
- 5.4.1 事件触发.mp4
- 5.4.2 事件切换.mp4
- 5.4.3 事件委派.mp4
- 5.4.4 事件命名空间.mp4
- 5.4.5 绑定多个事件.mp4
- 5.4.6 自定义事件.mp4
- 5.4.7 页面初始化事件.mp4
- 6.1.2 实例化XMLHttpRequest
- 6.2.1 建立连接.mp4
- 6.2.2 把对象转换为字符串
- 6.2.3 判断数组类型
- 6.2.4 判断函数类型
- 6.2.5 判断特殊对象
- 6.2.6 对数组和集合进行迭代
- 6.2.7 生成数组
- 6.2.8 对数组进行筛选
- 6.2.9 对数组进行转换
- 6.3.10 把多个数组合并在一起
- 6.3.11 删除数组中重复元素
- 6.3.12 在数组中查找指定值
- 7.1 简单的显示和隐藏
- 7.2 控制显示速度
- 7.2.3 显示切换
- 7.2.3 折叠动画
- 7.2.4 树形动画
- 7.2.5 选项卡动画
- 7.3.1 显示滑动效果
- 7.3.2 显示切换滑动
- 7.4.1 淡入和淡出
- 7.4.2 设置淡出透明效果
- 7.4.3 渐变切换
- 7.5.1 模拟show()效果
- 7.5.2 自定义动画
- 7.5.3 动态定位
- 7.5.4 停止动画
- 7.5.5 关闭动画
- 7.5.6 设置动画频率
- 7.5.7 延迟动画
- 7.6.1 添加动画队列
- 7.6.2 显示动画队列
- 7.6.3 更新动画队列
- 7.6.4 删除动画队列
- 8.1.1 检测用户代理
- 8.1.2 检测版本号
- 8.1.3 检测浏览器
- 8.1.4 检测功能或缺陷
- 8.2 兼容JavaScript库
- 8.3.1 处理字符串
- 8.3.2 把对象转换为字符串
- 8.3.3 判断数组类型
- 8.3.4 判断函数类型
- 8.3.5 判断特殊对象
- 8.3.6 对数组和集合进行迭代
- 8.3.7 生成数组
- 8.3.8 对数组进行筛选
- 8.3.9 对数组进行转换
- 8.3.10 把多个数组合并在一起
- 8.3.11 删除数组中重复元素
- 8.3.12 在数组中查找指定值
- 9.1.1 定义缓存
- 9.1.2 读取缓存
- 9.1.3 删除缓存
- 9.1.4 jQuery插件形式
- 9.1.5 自定义jQuery插件基本
- 9.1.6 使用extend()函数
- 9.1.7 自定义jQuery函数
- 9.1.8 自定义jQuery命令
- 9.1.9 自定义选择器
- 9.2.1 封装插件
- 9.2.2 优化插件
- 9.3.1 功能讲解
- 9.3.2 构建结构
- 9.3.3 设计思路
- 9.3.4 难点突破
- 9.3.5 代码实现
- 9.3.6 应用插件
- 10.1.1 服务器端分页
- 10.1.2 构建符合数据排序的表
- 10.1.3 实现表格基本排序
- 10.1.4 优化排序性能
- 10.1.5 设计其它类型排序
- 10.1.6 完善排序交互的视觉效果
- 10.2.1 快速过滤数据
- 10.2.2 处理多关键字匹配
- 10.2.3 处理特定列过滤
- 10.2.4 合成数据过滤器
- 10.2.5 快速编辑数据
- 10.2.6 完善数据编辑功能
- 11.1.1 设计表单结构
- 11.1.2 设计表单图标
- 11.1.3 设计提示信息
- 11.1.4 设计条件字段
- 11.2.1 验证服务概述

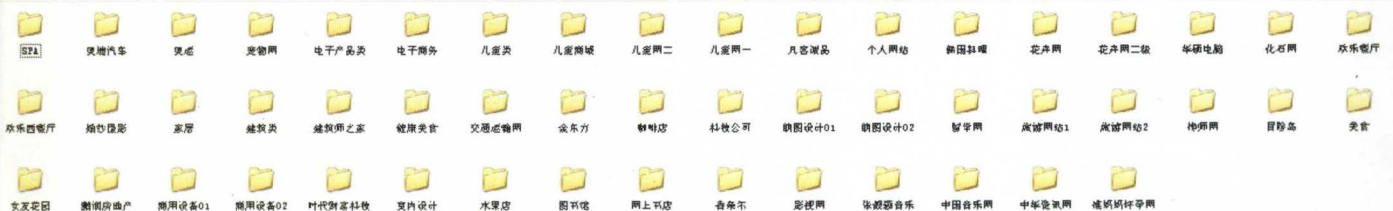
光盘 · 学习参考手册 (共11部)

- CSS 2.0 中文手册 已编译的 HTML 帮... 259 KB
- CSS 3.0参考手册 已编译的 HTML 帮... 361 KB
- JavaScript参考手册 已编译的 HTML 帮... 603 KB
- JavaScript核心参考手册 已编译的 HTML 帮...
- jQuery 1.7 中文手册 已编译的 HTML 帮... 298 KB
- jQuery 1.3参考手册 已编译的 HTML 帮... 94 KB
- jQuery 1.4参考手册 已编译的 HTML 帮... 490 KB
- W3CSchool 已编译的 HTML 帮... 5,862 KB
- w3c标准html5手册 已编译的 HTML 帮... 195 KB
- xHTML参考手册 已编译的 HTML 帮... 268 KB
- XMLHttpRequest中文参考手册 已编译的 HTML 帮... 24 KB

光盘 · HTML5+CSS3网页模版 (共32类)



光盘 · DIV + CSS3网页模版 (共51类)



光盘 · 各章节示例和源代码





清华社“视频大讲堂”大系
网络开发视频大讲堂

网络开发视频大讲堂

jQuery 开发从入门到精通

袁 江 编著

清华大学出版社

北 京

内 容 简 介

《jQuery 开发从入门到精通》(清华社“视频大讲堂”大系)通过基础知识+中小实例+综合案例的方式,讲述了 jQuery 入门,使用选择器,使用过滤器,DOM 操作,事件处理,Ajax 应用,动画设计,工具函数,功能扩展,表格开发,表单开发,jQuery UI 开发概述,jQuery UI 交互开发、部件开发、特效开发,jQuery 框架透析之函数式基础、面向对象基础、实战。相对 jQuery 权威指南,本书更能快速高效学习,学 jQuery mobile 者也可从本书获取一些基本知识。

本书还对 jQuery UI、jQuery 插件和实用工具函数等扩展知识,以及 jQuery 的开发技巧与性能优化等方面的重要知识做了详尽的阐述,以让读者轻松地使用 jQuery 来增强网页的互动性,做出更好的 Web 前端产品以及各种更炫更酷的效果。

本书显著特色有:

1. 同步视频讲解,让学习更为直观高效。327 节大型高清同步视频讲解,先看视频再学习效率更高。
2. 海量精彩实例,用实例学更轻松快捷。232 个精彩实例,模仿练习是最快捷的学习方式。
3. 精选实战案例,为高薪就业牵线搭桥。7 个实战案例展示可为以后就业积累经验。
4. 完整学习套餐,为读者提供贴心服务。学习参考手册 11 部,实用模版 83 类,素材源程序,让学习更加方便。
5. 讲解通俗翔实,看得懂学得会才是硬道理。

本书适合 Web 开发人员阅读和参考,同时也适合广大网页制作和设计的学生阅读和学习,也适合中高级用户进一步学习和参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

jQuery 开发从入门到精通/袁江编著. —北京:清华大学出版社,2013

(清华社“视频大讲堂”大系 网络开发视频大讲堂)

ISBN 978-7-302-30666-5

I. ①j… II. ①袁… III. ①Java 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 272540 号

责任编辑:赵洛育

封面设计:李志伟

版式设计:文森时代

责任校对:柴 燕

责任印制:刘海龙

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:清华大学印刷厂

装 订 者:北京市密云县京文制本装订厂

经 销:全国新华书店

开 本:203mm×260mm 印 张:39.5 字 数:1109 千字

(附 1DVD,含配套视频、参考手册、网页模板、素材源程序等)

版 次:2013 年 6 月第 1 版

印 次:2013 年 6 月第 1 次印刷

印 数:1~5000

定 价:79.80 元

产品编号:044321-01

功能丰富

前言

Preface

preface

jQuery 是功能丰富的 JavaScript 库，可以帮助用户毫不费力地把动态功能应用到网页。它的体积很小，代码风格独特而又优雅，改变了 JavaScript 程序员编写程序的方式和思路。jQuery 库有一个设计理念，那就是“写得少，做得多”（write less, do more），其独特的选择器、链式的 DOM 操作方式、事件绑定机制及封装完善的 Ajax 使其在众多优秀的 JavaScript 库中脱颖而出、独树一帜，赢得了众多使用者的拥护和信赖。

jQuery 的语法简单易学，而且具有很强大的跨平台性，可以兼容多种核心的浏览器。目前，已经有一百多个插件来扩充 jQuery 的功能，使得 jQuery 能满足几乎所有 客户端的脚本开发。

本书通过理论与实践相结合的方式，由浅入深、循序渐进地介绍了 jQuery 库的使用，同时又辅以大量真实的开发案例，让读者轻松使用 jQuery 来增强网页的互动性，做出更好的 Web 前端产品以及各种更炫更酷的效果。如果读者简单了解 HTML、CSS 和 JavaScript 基础知识，那么这本书正是为你而准备的，因为本书涵盖了利用 jQuery 展开工作时可能遇到的 大多数问题。

本书特色

□ 循序渐进，由浅入深

阅读本书不需要对 jQuery 有所了解，甚至对 JavaScript 也不需要有很深的了解。为了方便读者学习，本书系统地讲解了 jQuery 技术在网页设计中各个方面的应用知识，从为什么要用 jQuery 开始讲解，循序渐进，配合大量实例，帮助读者 奠定坚实的理论基础，做到 知其所以然，以期读者可以在不参考其他资料的情况下顺利过渡到 jQuery 的学习和使用。

□ 大量的案例实战

书中设置大量应用实例，重点强调具体技术的灵活应用，全书结合了作者长期的网页设计制作和教学经验，使读者真正做到学以致用。读者可以通过这些实例对 jQuery 的理论知识有更加深刻的理解，同时，这些实例稍作修改，就可以用在读者正在开发的项目中，实现各种精彩的效果。

□ 技术全面，内容充实

本书是关于 jQuery 的初级入门读物，书中详细介绍了 jQuery 1.6.4 几乎所有的特性和属性，并对每个模块均有很翔实的实例讲解，以期读者对 jQuery 有个很好的整体把握，同时以后需要用到一些特性时，可以查阅。另外，本书还在开始介绍了 Web 开发的基础知识，如 Web 开发中比较常用的工具等，可以让读者从一个完全的门外汉很快变成业内人士。

□ 图文结合，理解深刻

讲解技术类的知识，最好的方式就是面对面地讲授，但是图书却不太容易做到这一点。为了弥补这个缺憾，本书在讲解具体实例时，除了大量的注释、讲解之外，还辅以一些简洁明了的图片，以期让读者对实例以及 jQuery 效果有更直观的理解。

□ 配有源代码光盘，加速学习

为了让初学者快速入门，本书配套光盘中附赠了本书中大部分实例的源代码，读者可以参考阅读。但是，笔者依然强烈建议，在学习本书时应该边学边练，即便不能直接写代码，最好也要对着书上的代码手工敲入一遍，以加深印象和对知识本身的理解，在光盘中还有 HTML 参考手册、JavaScript 参考手册、CSS 参考手册、jQuery 参考手册等超值内容，在制作网页时也是很有用的参考。

本书内容

本书分为三大部分，共 18 章，具体结构划分如下。

第一部分：jQuery 基础知识部分，包括第 1~9 章。这部分主要介绍 jQuery 相关基础知识，包括 jQuery 相关概念、使用选择器、使用过滤器、DOM 操作、事件处理、Ajax 应用、动画设计、工具函数、功能扩展。

第二部分：应用开发部分，包括第 10~15 章。这部分主要介绍表格开发、表单开发、UI 开发、UI 交互开发、UI 部件开发、UI 特效开发。

第三部分：内核部分，包括第 16~18 章。这部分重点讲解 jQuery 内核构成和工作原理，主要包括 JavaScript 函数式基础、JavaScript 面向对象基础以及 jQuery 框架透析。

本书读者

- ❑ 希望系统学习网页设计、网站制作的初学者
- ❑ 从事网页设计制作和网站建设的专业人士
- ❑ 既适合初学者，也适合进阶者
- ❑ Web 前端开发和后台设计人员
- ❑ 可以作为各大中专院校相关专业的教学辅导和参考用书或相关培训机构的培训教材

本书约定

- ❑ 本书代码都以灰色背景显示，以方便读者阅读。考虑到版面限制，部分展示出来的代码仅包含 JavaScript 脚本和必要的结构代码。读者在学习测试时，应该把这些代码输入到网页。
- ❑ 本书以 jQuery 1.6.4 版本为基础进行介绍和演示，能够兼容 jQuery 3.0 以后的任何版本。
- ❑ 在默认情况下，jQuery 1.6.4 库文件都会自动导入文档，如果没有特别说明，我们会在示例中省略该行命令。jQuery 1.6.4 存放在 jQuery 文件夹中。
- ❑ 在默认情况下，使用 jQuery 的别名 \$ 来表示 jQuery 命名空间，同时直接把调用的函数放在 `$()` 函数中，该函数实际上是 `$("document").ready()` 方法的简写，它相当于 JavaScript 中的 `window.onload = function • () {}` 事件处理函数。
- ❑ 由于 jQuery 与 JavaScript 变量之间存在区别，默认情况下，当定义 jQuery 对象变量时，在变量的前面附加一个 \$ 前缀，以便与 JavaScript 变量区分。

关于我们

参与本书编写的人员包括咸建勋、奚晶、文菁、李静、钟世礼、李增辉、甘桂萍、刘燕、杨凡、李爱芝、余乐、孙宝良、余洪萍、谭贞军、孙爱荣、何子夜、赵美青、牛金鑫、孙玉静、左超红、蒋学军、邓才兵、袁江、李东博等。

由于作者水平有限，书中疏漏和不足之处在所难免，欢迎读者不吝赐教。广大读者如有好的建议、意见，或在学习本书时遇到疑难问题，可以联系我们，我们会尽快为您解答，服务邮箱为 design1993@163.com，liulm75@163.com。

编 者

估计花费是在500左右

不吝赐教

目 录




Contents

目录




内容





contents

第 1 章 初识 jQuery 1	2.2 层级选择器 27
 视频讲解: 43 分钟	2.2.1 包含选择器 28
1.1 jQuery 概述 1	2.2.2 子选择器 29
1.1.1 jQuery 能帮我做什么 1	2.2.3 相邻选择器 30
1.1.2 我需要学习 jQuery 2	2.2.4 兄弟选择器 31
1.2 使用 jQuery 5	2.2.5 层级选择器综合应用 32
1.2.1 下载 jQuery 6	2.2.6 解析层级选择器实现原理 34
1.2.2 安装 jQuery 6	2.3 简单的伪类选择器 37
1.2.3 测试 jQuery 7	2.3.1 特定位置选择器 37
1.3 jQuery 框架的优势 8	2.3.2 指定范围选择器 39
1.4 jQuery 框架核心功能 9	2.3.3 排除选择器 40
1.4.1 对外接口单一让使用更简单 9	2.3.4 特殊选择器 41
1.4.2 链式语法让编码更快速、优雅 9	2.3.5 解析简单伪类选择器的实现原理 42
1.4.3 模仿 CSS 选择器让选取元素 更精确、灵活 11	2.4 与内容相关的伪类选择器 42
1.4.4 扩展接口让 jQuery 更开放、富有活力 13	2.4.1 匹配包含文本选择器 43
1.5 初学 jQuery 最容易混淆的几个概念 14	2.4.2 匹配包含元素选择器 44
1.5.1 jQuery 对象不等于 DOM 对象 14	2.4.3 包含判断选择器 45
1.5.2 jQuery 对象与 DOM 对象之间的转换 15	2.4.4 解析内容过滤器实现原理 45
1.5.3 jQuery 的 ready 不等于 JavaScript 的 load 17	2.5 与元素显示状态相关的伪类选择器 46
1.6 学习资源 19	2.6 匹配子元素的伪类选择器 47
1.6.1 jQuery 开发工具 19	2.7 与表单对象相关的伪类选择器 49
1.6.2 jQuery 参考手册 19	2.8 与表单属性相关的伪类选择器 53
1.6.3 jQuery 在线资源 20	2.9 属性选择器 54
第 2 章 使用选择器 21	2.10 jQuery 选择器应用优化 58
 视频讲解: 1 小时 28 分钟	第 3 章 使用过滤器 61
2.1 基本选择器 21	 视频讲解: 55 分钟
2.1.1 ID 选择器 21	3.1 过滤 61
2.1.2 标签选择器 23	3.1.1 类过滤 61
2.1.3 类选择器 24	3.1.2 下标过滤 63
2.1.4 通配选择器 25	3.1.3 表达式过滤 64
2.1.5 组选择器 26	3.1.4 判断 67
	3.1.5 映射 67
	3.1.6 清洗 69





3.1.7 截取	69	4.8.3 切换类样式	122
3.2 查找	70	4.8.4 判断样式	124
3.2.1 向下查找后代元素	71	4.9 读写文本和值	124
3.2.2 向上查找祖先元素	73	4.9.1 读写 HTML	124
3.2.3 向上查找兄弟元素	78	4.9.2 读写文本	126
3.2.4 向下查找兄弟元素	80	4.9.3 读写值	127
3.2.5 查找兄弟元素	83	4.10 样式表操作	129
3.2.6 添加查找对象	84	4.10.1 读写 CSS 样式	129
3.3 串联	84	4.10.2 绝对定位	133
3.3.1 绑定前后 jQuery 对象	85	4.10.3 相对定位	134
3.3.2 返回前一个 jQuery 对象	86	4.10.4 设置大小	136
第 4 章 DOM 操作	87	4.11 访问文档树	137
 视频讲解: 2 小时 10 分钟		第 5 章 事件处理	140
4.1 创建节点	88	 视频讲解: 1 小时 30 分钟	
4.1.1 创建元素	89	5.1 事件处理模型	140
4.1.2 输入文本	90	5.1.1 原始事件模型	140
4.1.3 设置属性	90	5.1.2 DOM 事件模型	141
4.2 插入内容	92	5.1.3 IE 事件模型	144
4.2.1 内部插入	92	5.2 事件处理机制	146
4.2.2 外部插入	96	5.2.1 Event 对象	147
4.3 删除内容	98	5.2.2 事件流	148
4.3.1 移出	99	5.2.3 事件控制	151
4.3.2 清空	101	5.3 jQuery 事件封装机制	151
4.3.3 分离	101	5.3.1 注册事件	152
4.4 克隆内容	103	5.3.2 注销事件	155
4.5 替换内容	105	5.4 jQuery 事件应用	157
4.6 包裹内容	107	5.4.1 事件触发	158
4.6.1 外包	107	5.4.2 事件切换	159
4.6.2 内包	108	5.4.3 事件委派	163
4.6.3 总包	110	5.4.4 事件命名空间	165
4.6.4 卸包	111	5.4.5 绑定多个事件	167
4.7 属性操作	111	5.4.6 自定义事件	168
4.7.1 设置属性	112	5.4.7 页面初始化事件	169
4.7.2 访问属性	115	第 6 章 Ajax 应用	172
4.7.3 删除属性	118	 视频讲解: 1 小时 12 分钟	
4.8 类操作	120	6.1 XMLHttpRequest 基础	172
4.8.1 添加类样式	120	6.1.1 XMLHttpRequest 对象	172
4.8.2 删除类样式	121		


目 录

6.1.2 实例化 XMLHttpRequest	173	7.6 动画队列	225
6.1.3 建立连接	174	7.6.1 添加动画队列	225
6.1.4 请求和响应	175	7.6.2 显示动画队列	227
6.2 jQuery Ajax	177	7.6.3 更新动画队列	227
6.2.1 设计一个简单的示例	178	7.6.4 删除动画队列	229
6.2.2 GET 请求	179	第 8 章 工具函数	230
6.2.3 POST 请求	182	 视频讲解: 1 小时 21 分钟	
6.2.4 ajax() 方法请求	184	8.1 jQuery 标志	230
6.2.5 响应状态	186	8.1.1 检测用户代理	230
6.2.6 响应信息	188	8.1.2 检测版本号	233
6.2.7 载入网页文件	190	8.1.3 检测盒模型	234
6.2.8 预设参数项	192	8.1.4 检测功能或缺陷	234
6.2.9 预处理字符串	193	8.2 兼容 JavaScript 库	235
第 7 章 动画设计	196	8.3 对象和集合操作	238
 视频讲解: 1 小时 23 分钟		8.3.1 处理字符串	238
7.1 CSS 动画设计基础	196	8.3.2 把对象转换为字符串	239
7.2 显隐动画	198	8.3.3 判断数组类型	240
7.2.1 简单的显示和隐藏	199	8.3.4 判断函数类型	241
7.2.2 控制显示速度	200	8.3.5 判断特殊对象	242
7.2.3 显隐切换	201	8.3.6 对数组和集合进行迭代	243
7.2.4 折叠动画	202	8.3.7 生成数组	245
7.2.5 树形动画	204	8.3.8 对数组进行筛选	246
7.2.6 选项卡动画	207	8.3.9 对数组进行转换	247
7.3 滑动动画	208	8.3.10 把多个数组合并在一起	249
7.3.1 显隐滑动效果	209	8.3.11 删除数组中重复元素	251
7.3.2 显隐切换滑动	210	8.3.12 在数组中查找指定值	252
7.4 渐变效果	212	8.4 缓存	252
7.4.1 淡入和淡出	212	8.4.1 定义缓存	253
7.4.2 设置淡出透明效果	214	8.4.2 读取缓存	255
7.4.3 渐变切换	215	8.4.3 删除缓存	256
7.5 复杂动画	216	第 9 章 功能扩展	259
7.5.1 模拟 show() 方法的效果	217	 视频讲解: 1 小时 32 分钟	
7.5.2 自定义动画	218	9.1 自定义插件	259
7.5.3 动态定位	221	9.1.1 jQuery 插件形式	259
7.5.4 停止动画	222	9.1.2 自定义 jQuery 插件基本规则	260
7.5.5 关闭动画	223	9.1.3 使用 extend() 函数	261
7.5.6 设置动画频率	224	9.1.4 自定义 jQuery 函数	265
7.5.7 延迟动画	225	9.1.5 自定义 jQuery 命令	267

9.1.6 自定义选择器.....	271	11.1.3 设计提示信息.....	321
9.2 封装和优化插件.....	275	11.1.4 设计条件字段.....	322
9.2.1 封装插件.....	275	11.2 表单验证.....	324
9.2.2 优化插件.....	277	11.2.1 验证服务概述.....	324
9.3 案例实战：制作 jQuery 文字提示 插件.....	283	11.2.2 认识正则表达式.....	325
9.3.1 功能讲解.....	283	11.2.3 字符匹配.....	327
9.3.2 构建结构.....	283	11.2.4 重复匹配.....	331
9.3.3 设计思路.....	284	11.2.5 高级匹配.....	335
9.3.4 难点突破.....	286	11.2.6 匹配操作.....	342
9.3.5 代码实现.....	287	11.2.7 联系表单验证.....	347
9.3.6 应用插件.....	290	11.3 增强型表单.....	350
第 10 章 表格开发.....	291	11.3.1 自适应多行文本框.....	350
 视频讲解：1 小时 10 分钟		11.3.2 注册码文本框.....	352
10.1 数据排序.....	291	11.3.3 掩码输入文本框.....	357
10.1.1 构建符合数据排序的表格结构.....	291	第 12 章 jQuery UI 开发概述.....	362
10.1.2 JavaScript 的基本排序方法.....	293	 视频讲解：49 分钟	
10.1.3 实现表格基本排序.....	296	12.1 jQuery UI 开发.....	362
10.1.4 优化排序性能.....	298	12.1.1 设计思想.....	362
10.1.5 设计其他类型排序.....	299	12.1.2 设计体验.....	366
10.1.6 完善排序交互的视觉效果.....	301	12.2 使用 jQuery UI 库.....	372
10.2 数据分页.....	303	12.2.1 认识 jQuery 插件库.....	373
10.2.1 服务器端分页.....	303	12.2.2 使用外部插件.....	373
10.2.2 JavaScript 实现分页.....	307	12.2.3 认识 UI 插件.....	376
10.3 数据过滤.....	310	12.2.4 建立开发环境.....	377
10.3.1 快速过滤数据.....	310	12.2.5 jQuery UI 库结构.....	377
10.3.2 处理多关键字匹配.....	311	12.2.6 主题定制器.....	378
10.3.3 处理特定列过滤.....	311	12.2.7 如何使用 jQuery UI 组件.....	379
10.3.4 合成数据过滤器.....	312	12.2.8 组件类别.....	379
10.4 数据编辑.....	314	12.2.9 浏览器支持.....	380
10.4.1 快速编辑数据.....	314	第 13 章 jQuery UI 交互开发.....	381
10.4.2 完善数据编辑功能.....	315	 视频讲解：40 分钟	
第 11 章 表单开发.....	318	13.1 拖放.....	381
 视频讲解：2 小时 9 分钟		13.1.1 拖动对象.....	382
11.1 设计可用性表单.....	318	13.1.2 投放对象.....	388
11.1.1 设计表单结构.....	318	13.2 缩放.....	392
11.1.2 设计表单图标.....	320	13.3 选择.....	396
		13.4 排序.....	400


目 录

第 14 章 jQuery UI 部件开发.....	405	16.4.2 this 对象	468
 视频讲解: 51 分钟		16.4.3 this 应用	471
14.1 选项卡	405	16.4.4 this 陷阱	474
14.2 手风琴	410	16.5 动态调用	478
14.3 对话框	415	16.6 函数作用域	481
14.4 滑动条	420	16.6.1 词法作用域与执行作用域	482
14.5 日期选择器	425	16.6.2 作用域链	484
第 15 章 jQuery UI 特效开发.....	431	16.6.3 调用对象	485
 视频讲解: 49 分钟		16.7 闭包函数	486
15.1 特效核心	432	16.7.1 认识闭包	487
15.2 高亮	434	16.7.2 闭包基本特性	488
15.3 弹跳	436	16.7.3 闭包基本用法	489
15.4 摇晃	439	16.7.4 闭包标识系统	491
15.5 转换	440	16.7.5 闭包函数作用域	492
15.6 缩放	441	16.7.6 闭包函数生存周期	494
15.7 爆炸	442	16.7.7 比较函数和闭包	495
15.8 抖动	444	16.7.8 闭包函数与函数实例	496
15.9 落体	445	16.7.9 闭包函数和调用对象	499
15.10 滑动	447	16.7.10 闭包独立性	501
15.11 剪辑	448	16.7.11 构造函数闭包	502
15.12 百叶窗	449	16.7.12 应用闭包函数	503
15.13 折叠	450	16.7.13 闭包副作用	507
第 16 章 jQuery 框架透析之函数式基础.....	452	第 17 章 jQuery 框架透析之面向	
 视频讲解: 2 小时 40 分钟		对象基础	510
16.1 定义函数	453	 视频讲解: 4 小时 44 分钟	
16.1.1 构造函数	453	17.1 定义对象	510
16.1.2 函数直接量	455	17.1.1 认识对象	511
16.1.3 选择恰当的方法	456	17.1.2 定义对象	512
16.2 使用函数	458	17.2 使用对象	513
16.2.1 函数调用	458	17.2.1 引用对象	513
16.2.2 生命周期	459	17.2.2 销毁对象	514
16.2.3 形参和实参	460	17.2.3 定义对象属性	514
16.2.4 参数对象 Arguments	461	17.2.4 访问对象属性	515
16.2.5 回调函数 callee	463	17.2.5 操作对象属性	515
16.2.6 返回值	463	17.2.6 操作对象方法	516
16.3 函数对象	464	17.3 对象作用域	517
16.4 动态指针	466	17.3.1 公共作用域	517
16.4.1 认识 this	466	17.3.2 私有作用域	518

17.3.3 静态作用域.....	518	17.11 封装.....	570
17.3.4 对象指针 this.....	518	17.11.1 被动封装.....	570
17.4 对象类型.....	519	17.11.2 主动封装.....	571
17.4.1 构造对象.....	519	17.11.3 静态方法.....	573
17.4.2 实例对象.....	521	17.12 重载和多态.....	575
17.4.3 原型对象.....	521	17.12.1 重载.....	575
17.4.4 构造器 constructor.....	524	17.12.2 覆盖.....	575
17.5 核心方法.....	525	17.12.3 多态.....	576
17.5.1 toString()方法.....	526	17.13 构造和析构.....	577
17.5.2 valueOf()方法.....	527	17.13.1 构造.....	577
17.5.3 hasOwnProperty()方法.....	528	17.13.2 析构.....	579
17.5.4 propertyIsEnumerable()方法.....	528	17.14 扩展.....	580
17.5.5 isPrototypeOf()方法.....	530	17.14.1 超类和子类.....	580
17.6 核心对象.....	530	17.14.2 元类.....	581
17.6.1 对象系统.....	531	第 18 章 jQuery 框架透析之实战.....	583
17.6.2 Global 对象.....	531	 视频讲解: 1 小时 52 分钟	
17.6.3 Math 对象.....	532	18.1 设计思路.....	583
17.6.4 Date 对象.....	534	18.2 设计框架.....	584
17.7 类型.....	535	18.2.1 定义构造函数.....	585
17.7.1 认识类.....	535	18.2.2 返回 jQuery 对象.....	586
17.7.2 定义类.....	536	18.2.3 设计作用域.....	587
17.8 接口.....	541	18.2.4 跨域访问.....	588
17.8.1 认识接口.....	541	18.2.5 设计选择器.....	589
17.8.2 定义接口.....	542	18.2.6 设计迭代器.....	590
17.9 原型.....	546	18.2.7 设计扩展接口.....	592
17.9.1 认识 prototype.....	546	18.2.8 解决参数传递问题.....	594
17.9.2 原型特性.....	548	18.2.9 设计名字空间.....	595
17.9.3 原型操作.....	550	18.3 构建 jQuery 对象.....	596
17.9.4 定义静态原型.....	554	18.4 构建 jQuery DOM 元素.....	599
17.9.5 原型域和原型域链.....	554	18.4.1 生成 DOM 元素.....	599
17.10 继承.....	556	18.4.2 间接引用 DOM 节点.....	602
17.10.1 原型继承.....	556	18.4.3 采用 CSS 方式查找 DOM 节点.....	604
17.10.2 类继承(上).....	557	18.5 类数组.....	607
17.10.3 类继承(下).....	561	18.5.1 构建类数组.....	607
17.10.4 实例继承.....	563	18.5.2 操作类数组.....	608
17.10.5 复制继承.....	564	18.6 Sizzle 引擎.....	612
17.10.6 克隆继承.....	565	18.6.1 设计思路.....	612
17.10.7 混合继承.....	565	18.6.2 设计框架.....	614
17.10.8 多重继承.....	566		

第 1 章

初识 jQuery

( 视频讲解：43 分钟)

jQuery 是一个 JavaScript 代码仓库，开发人员习惯称之为 JavaScript 框架，它可以帮助用户使用很少的 JavaScript 代码，创建出漂亮的页面效果。jQuery 设计的宗旨就是“Write Less, Do More”，即倡导写更少的代码，做更多的事情。

1.1 jQuery 概述

jQuery 诞生于 2005 年，由 John Resign 开发，如图 1.1 所示。到现在，jQuery 经历了 7 年的时间洗涤，成为全球最受欢迎的 JavaScript 框架之一。目前，微软的 Visual Studio 2008+ 和 Adobe 的 Dreamweaver CS 5.5+ 都完全包含了 jQuery 框架，并实现核心支持和扩展。



图 1.1 jQuery 框架的作者 John Resign

1.1.1 jQuery 能帮我做什么

jQuery 功能很强大，在客户端开发中，它主要帮助用户方便、快速地完成以下任务：

☒ 选择页面元素

如果不使用 jQuery，直接使用 JavaScript 遍历 DOM (Document Object Model, 文档对象模型) 树，以及查找 HTML 文档结构中的某个元素，必须编写很多行代码。在 jQuery 中提供了可靠而富有效率的选择器，只需要一个简单的选择器字符串，即可准确获取需要检查或操纵的文档元素。

☒ 动态更改页面样式

使用 JavaScript 控制 CSS 受限于不同浏览器的兼容性处理，而 jQuery 可以弥补这一不足，它提供了浏览器的标准解决方案。而且即使在页面已经呈现之后，jQuery 仍然能够改变文档中某个部分的类或者个别样式属性。

☒ 动态更改页面内容

jQuery 能改变文档的内容。使用少量的代码，即可改变网页内容，对 HTML 文档的整个结构都能重写或者扩展。使用起来远比 JavaScript 直接控制便捷。

选择页面元素

☑ 控制响应事件

jQuery 提供了丰富的页面事件，这些事件使用简单、易用、易记，不需要考虑浏览器兼容性问题，但是如果使用 JavaScript 直接控制用户行为，需要考虑的问题就很多，既要考虑 HTML 文档结构与事件处理函数的合成，还要考虑浏览器的不一致性。

☑ 提供基本网页特效

jQuery 内置了一批淡入、擦除、移动之类的效果，以及制作新效果的工具包，用户只需要简单地调用动画函数，就可以快速设计出高级动画效果。如果直接使用 JavaScript 实现，需要考虑 CSS 动态控制，还要顾虑浏览器解析差异，模拟的动画效果或许很生硬，或者很粗糙等。

☑ 快速实现通信

jQuery 对 Ajax 技术的支持很缜密，它通过消除这一过程中浏览器特定的复杂性，使用户得以专注于服务器端的功能设计。

☑ 扩展 JavaScript 内核

jQuery 提供了对 JavaScript 核心功能的扩展，如迭代和数组操作等，增加对客户端、数据存储及 JavaScript 扩展的支持。

1.1.2 我需要学习 jQuery

随着近年来互联网对用户体验的重视，催生了客户端开发的热潮，由此也捧红了一大批 JavaScript 框架。互联网上产生大量的 JavaScript 框架，有的专注于上述任务中的一项或两项，有的则试图以预打包的形式囊括各种可能的行为和动态效果。例如，下面这些 JavaScript 框架都比较有名，在这些框架中可能有你需要的部分功能或者选项，或者完整的解决方案。

1. Dojo

Dojo (<http://dojotoolkit.org/>) 是一个强大的面向对象的 JavaScript 框架，如图 1.2 所示。它主要由三大模块组成：Core、Dijit、DojoX。Core 提供 Ajax、events、packaging、CSS-based querying、animations、JSON 等相关操作 API。Dijit 是一个可更换皮肤、基于模板的 Web UI 控件库。DojoX 包括一些创新/新颖的代码和控件，如 DateGrid、charts、离线应用和跨浏览器矢量绘图等。



图 1.2 Dojo 官网

2. YUI

YUI (<http://developer.yahoo.com/yui/>) 是 Yahoo! User Interface Librar 库的简称，它是一组采用 DOM

Scripting、DHTML 和 Ajax 等技术开发的 Web UI 控件和工具。中文翻译就是 Yahoo 用户界面库，如图 1.3 所示。

- ☑ YUI 工具包利用 DOM 脚本来简化浏览器内的开发(in-browser devolvement)，使用 DHTML 和 Ajax 的特性开发所有的 Web 程序。
- ☑ YUI 控件库为页面提供一组高交互性的可视化元素。这些元素完全在客户端创建和维护，不需要请求服务器进行页面刷新。

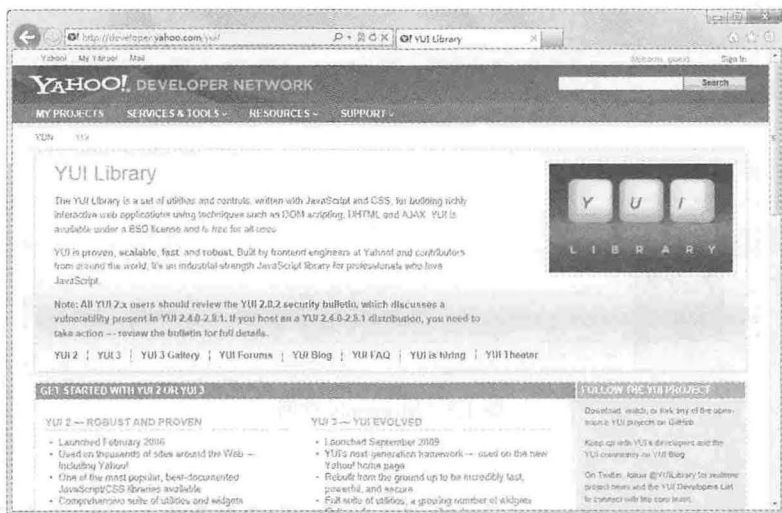


图 1.3 YUI 官网

3. jQuery

jQuery (<http://jquery.com/>) 是一个快速、简洁的 JavaScript 框架，可以简化查询 DOM 对象、处理事件、制作动画、处理 Ajax 交互的过程。利用 jQuery 将改变编写 JavaScript 代码的方式。如果使用 JavaScript 需要 20 行代码完成的功能，jQuery 只需要 5 行就可以轻松搞定，如图 1.4 所示。



图 1.4 jQuery 官网

4. Mootools

Mootools (<http://mootools.net/>) 是一个简洁、模块化、面向对象的 JavaScript 框架。它能够更快、更简单地编写可扩展和兼容性强的 JavaScript 代码。Mootools 从 Prototype 中汲取了许多有益的设计理念，语法

与 Prototype 极其类似。但它提供的功能要比 Prototype 多，整体设计也比 Prototype 要相对完善，功能更强大，增加了动画特效、拖放操作等，如图 1.5 所示。

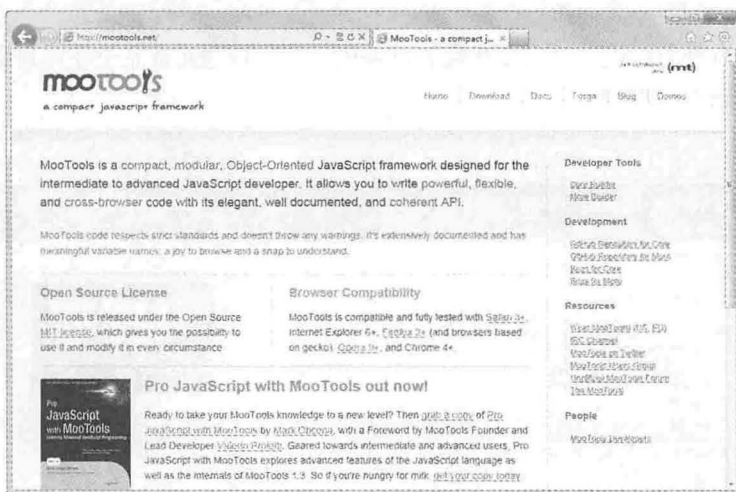


图 1.5 MooTools 官网

5. Prototype

Prototype (<http://www.prototypejs.org/>) 是一个易于使用、面向对象的 JavaScript 框架。它封装并简化和扩展一些在 Web 开发过程中常用到的 JavaScript 方法与 Ajax 交互处理过程，如图 1.6 所示。

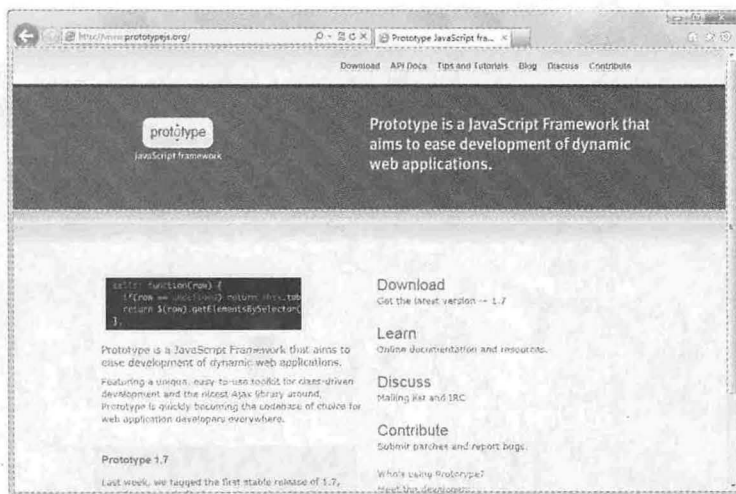


图 1.6 Prototype 官网

6. ExtJS

ExtJS (<http://www.sencha.com/>) 是一个跨浏览器，用于开发 RIA (Rich iNternet Application) 应用的 JavaScript 框架，提供高性能、可定制的 Web UI 控件库。该框架具有良好的设计、丰富的文档和可扩展的组件模型，如图 1.7 所示。

但是，与其他优秀框架相比，jQuery 在下面几个方面更胜一筹。

☒ 沿袭 CSS 选择符用法，简化选择操作

通过将查找页面元素的机制构建于 CSS 选择符之上，jQuery 继承了简明清晰地表达文档结构的方式。由于大部分用户对于 CSS 语法比较熟悉，因而使用 jQuery 就更容易上手。

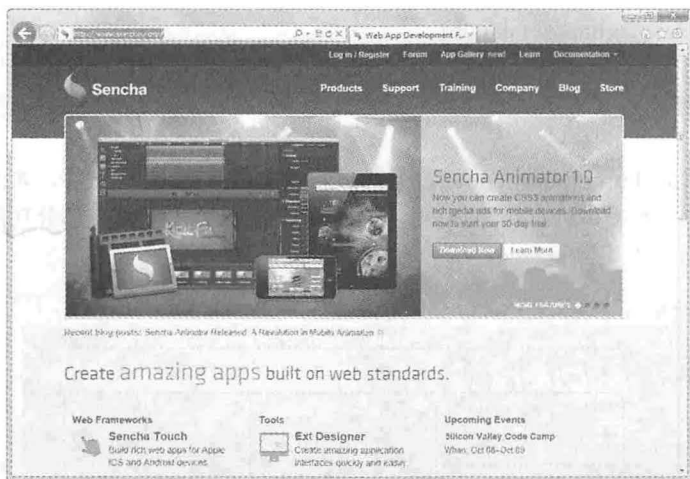


图 1.7 ExtJS 官网

☑ 无限制扩展

再强大的 JavaScript 特效库，都无法满足用户的个性化需求，jQuery 通过提供简单、统一的扩展接口以满足众口难调的难题。创建新插件的方法很简单，而且拥有完备的文档说明，这促进了大量有创意和有实用价值的模块的开发。甚至在下载的基本 jQuery 库文件当中，多数特性在内部都是通过插件架构实现的。而且，如有必要，可以移除这些内部插件，从而生成更小的库文件。

☑ 兼容浏览器

Web 开发领域中一个令人遗憾的事实是，每种浏览器对颁布的标准都有自己的一套不一致的实现方案。任何 Web 应用程序中都会包含一个用于处理这些平台间特性差异的重要组成部分。虽然不断发展的浏览器前景为某些高级特性提供浏览器中立的完美的基础代码（code base）不大可能，但 jQuery 添加一个抽象层来标准化常见的任务，从而有效地减少了代码量，同时，也极大地简化了这些任务。

☑ 集合化操作

jQuery 选择 DOM 元素之后，会自动封装成一个集合对象（也称为伪数组），调用 jQuery 方法可以直接对这些集合元素进行操作，而不需要循环遍历每一个返回的元素。相反，.hide()之类的方法被设计成自动操作对象集合，而不是单独的对象。这种称作隐式迭代的技术，使得大量的循环结构变得不再必要，从而大幅地减少代码量。

☑ 优化代码书写，提高开发效率

借助链式语法，jQuery 将多重操作集于一行。为了避免过度使用临时变量或不必要的重复代码，jQuery 在其多数方法中采用了链式编程模式。这种模式意味着基于一个对象进行的大多数操作的结果，都会返回该对象本身，以便于为该对象应用下一次操作。

jQuery 小巧、便捷，学习门槛比较低，并且为用户使用这个库的自定义代码保持简洁提供了技术保障。

1.2 使用 jQuery

jQuery 项目主要包括 jQuery Core（核心库）、jQuery UI（界面库）、Sizzle（CSS 选择器）和 QUnit（测试套件）4 部分，团队成员由核心库开发、UI 开发、插件开发，以及网站设计、运营和推广人员组成。几个主要网站的网址如下：

☑ jQuery 框架官网 <http://jquery.com/>。

☑ jQuery 项目组官网 <http://jquery.org/>。

☒ John Resign 个人网站 <http://ejohn.org/>。

1.2.1 下载 jQuery

访问 jQuery 官方网站 (<http://jquery.com/>)，下载最新版本的 jQuery 库文件，如图 1.8 所示。目前最新版本是 1.6.4。本书主要根据 1.6 版本进行讲解，由于 jQuery 1.6 仅增加和完善了部分功能，所以读者使用 jQuery 1.3+ 版本就可以进行学习和上机测试。



图 1.8 下载 jQuery 最新版本

在 jQuery 官网首页右侧，先选中要下载的 jQuery 库文件类型，主要包括发布版和测试版两种：

- ☒ PRODUCTION (31KB, Minified and Gzipped)
- ☒ DEVELOPMENT (229KB, Uncompressed Code)

如果选择 PRODUCTION（发布）选项，则可以下载代码压缩版本，此时 jQuery 框架源代码被压缩到了 31KB，下载的文件为 `jquery-1.6.4.min.js`。如果选择 DEVELOPMENT（开发）选项，则可以下载包含注释的未被压缩的版本，大小为 233KB，下载的文件为 `jquery-1.6.4.js`。然后，单击 `Download(jQuery)` 按钮即可。

如果需要 jQuery 其他版本的源代码，可以访问下面网址进行下载：

- ☒ <http://github.com/jquery/jquery>
- ☒ <http://code.google.com/p/jqueryjs/>
- ☒ http://docs.jquery.com/Downloading_jQuery

当然 jQuery 官方网站始终都包含与该库有关的最新代码和资讯。为了方便学习，需要从官方网站上下载一个 jQuery 库文件。官方网站在任何时候都会提供几种不同版本的 jQuery 库，但其中最适合读者学习的是该库最新的未压缩版。

1.2.2 安装 jQuery

使用 jQuery 库不需要安装，只要把下载的库文件放到站点中即可。因为 JavaScript 是一种解释型语言，所以使用它不必进行编译或者构建。无论什么时候，当我们在某个页面上使用 jQuery 时，只需要在 HTML 文档头部简单地引用该库文件即可。

【示例 1】 导入 jQuery 库文件可以使用相对路径，也可以使用绝对路径，具体情况根据读者存放 jQuery 库文件的位置而定。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
    //在这里用户就可以使用 jQuery 编程了
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

注意，除了可以使用上述方法将 jQuery 库导入页面中外，还可以使用 Google 在线提供的库文件。在大多数环境下，推荐使用 Google 提供的 jQuery 代码，因为使用 Google 存储的 jQuery 更加稳定、可靠、高速。

```

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.4/jquery.min.js"></script>

```

或者使用 Google 提供的 API 进行导入：

```

<script type="text/javascript" src="http://www.google.com/jsapi"></script>
<script type="text/javascript">
google.load("jquery", "1.6.4", {uncompressed:true});
</script>

```

google.load()函数包含 3 个参数，第 1 个参数为 JavaScript 库的名称，如 jquery、extjs 等；第 2 个参数为该库的版本号，如 1.6.4；第 3 个参数设定是否使用压缩版本的库文件，使用未压缩版本格式为 {uncompressed:true}。前两个参数必选，第 3 个参数为可选。

提示，Google Ajax Libraries API 是 Google 的一个项目，它提供当前流行的各种 JavaScript 库的快速引用方式，并承诺永久可用。

1.2.3 测试 jQuery

引入 jQuery 库文件之后，用户就可以进行 jQuery 开发了。开发的步骤很简单，在导入 jQuery 库文件的 <script> 标签行下面，重新使用 <script> 标签定义一个 JavaScript 代码段，然后就可以在 <script> 标签内调用 jQuery 方法进行 Web 开发了。

【示例 2】 在页面初始化完毕后，通过 JavaScript 的 alert() 方法与浏览者打个招呼。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    alert("Hi,您好!");
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

在浏览器中预览该网页文件，则可以看到在当前窗口中会弹出一个提示对话框，如图 1.9 所示。



图 1.9 测试 jQuery 结果

【代码详解】

在 jQuery 库中，\$ 是 jQuery 的别名，如 `$()` 等效于 `jQuery()`。jQuery() 函数是 jQuery 库文件的接口函数，所有 jQuery 操作都必须从该接口函数切入。jQuery() 函数相当于页面初始化事件处理函数，当页面加载完毕，会执行 jQuery() 函数包含的函数，所以当浏览该页面时，会执行 “`alert("Hi,您好!");`” 代码，然后将看到弹出的信息提示对话框。

注意，使用 jQuery 进行开发时，如果在做所有事情之前，希望 jQuery 操作 DOM 文档，则必须确保在 DOM 载入完毕后开始执行，应该使用 ready 事件作为处理 HTML 文档的开始。

```
$(document).ready(function() {
    //JavaScript 或者 jQuery 代码
});
```

上面代码的语义是，匹配文档中的 document 节点，然后为该节点绑定 ready 事件处理函数。它类似于 JavaScript 的 window.onload 事件处理函数，不过 jQuery 的 ready 事件要先于 onload 事件被激活。

```
window.onload = function(){
    //JavaScript 或者 jQuery 代码
};
```

为了方便开发，jQuery 框架进一步简化了 `$(document).ready()` 方法的写法，直接使用 `$()` 方法来表示。

```
$( function() {
    //JavaScript 或者 jQuery 代码
});
```

考虑到页面加载需要一个过程，所有 jQuery 代码建议都包含在 `$()` 函数中，当然也可以不被包含在 `$()` 函数中，这与 JavaScript 代码应该放在 window.onload 事件处理函数中的道理是一样的。

1.3 jQuery 框架的优势

jQuery 为用户提供了解决跨浏览器兼容方案，使 DOM 操作趋于统一，jQuery 确保代码能够跨越所有主要浏览器以一致的方式进行工作，摆在了高优先级的位置，许多让开发人员头疼的事件处理、样式设计等兼容操作问题变得轻松、方便许多。简单概括，jQuery 框架具有下面几个优势：

- ☑ 体积小，使用灵巧。
- ☑ 丰富的 DOM 选择器（CSS 1~3、XPath）。
- ☑ 跨浏览器（IE 6、FF、Safari、Opera）。
- ☑ 链式代码。
- ☑ 强大的事件、样式支持。
- ☑ 强大的 Ajax 功能。
- ☑ 易于扩展、插件丰富。

1.4 jQuery 框架核心功能

jQuery 框架的核心就是从 HTML 文档中匹配元素并对其执行操作。如果读者熟悉 CSS 技术,就会对 CSS 选择器的强大威力有所感触。CSS 选择器能够通过元素的特性或者文档位置来描述元素的样式,JavaScript 还无法内置 CSS 选择器的功能,仅提供两个有限的选择 DOM 元素的方法。现在 jQuery 通过封装 JavaScript 的原生方法,模拟了一套 CSS 选择器,甚至定义了完整的 XPath 语言的选择能力,这在一定程度上简化了 JavaScript 的操作。

1.4.1 对外接口单一让使用更简单

jQuery 把所有的操作都封装在一个 jQuery() 函数中,形成统一(也是唯一)的操作入口,这为 jQuery 操作降低了门槛。jQuery() 构造函数能够接收任意类型的数据,但是能够解析的参数主要有以下 4 种类型。

1. jQuery(expression, context)

参数 expression 为一个表达式,该表达式可以是 ID、DOM 元素名、CSS 表达式、XPath 表达式等, jQuery 将根据表达式匹配文档中的元素,然后把找到的元素包装到一个 jQuery 对象中返回。例如:

```
jQuery("div#wrap>p:first").addClass("red");
```

在表达式字符串中,div#wrap 表示 id 为 wrap 的 div 元素,然后在该元素中匹配子元素 p,最后筛选出第 1 个 p 元素。"div#wrap>p:first" 是 CSS 表达式,如果使用 XPath 表达式表示,则应该为"div#wrap/p:first",:first 是一个伪类,表示其中的第 1 个。addClass() 为 jQuery 对象用来添加 CSS 样式类的方法,相反操作的方法为 removeClass()。

2. jQuery(html)

参数 html 表示一个 HTML 结构的字符串,此时 jQuery 将创建一个对应结构的 HTML 文档片段。例如:

```
$('#ul').append($('
```

\$('#li>new item') 将其中的字符串转换为 DOM 对象,然后通过 append() 方法加入 ul 元素的最后。

3. jQuery(elements)

参数 elements 是一个 DOM 元素对象或者集合,此时 jQuery 将把 DOM 元素或集合中的 DOM 元素封装为 jQuery 对象。例如:

```
$(document).ready(function(){  
    $('#ul').css('color','red');  
});
```

其中, jQuery 构造函数把 document 对象封装为一个 jQuery 对象,然后调用 ready() 方法。ready 事件处理函数为 document 对象绑定一个事件,当页面初始化后,将 ul 的颜色设置为红色。

4. jQuery(fn)

参数 fn 是一个处理函数。\$(document).ready() 由于使用频繁,所以 jQuery 又使用 \$(fn) 来代替,fn 表示处理函数。例如:

```
$(function(){  
    $('#ul').css('color','red');  
});
```

1.4.2 链式语法让编码更快速、优雅

jQuery 的代码是非常优雅,也是非常灵巧的。它允许用户连续编写各种行为,从而实现按人的惯性思

维进行快速开发，这种代码形式被称为链式语法。

【示例 3】 仅写了两行脚本，即实现复杂的页面交互效果，如图 1.10 所示。第 1 行代码使用 jQuery 构造函数 (\$) 创建 4 个按钮，并把它们附加到文档中。第 2 行代码通过链式语法添加连续的行为，分别选中这 4 个按钮并为它们绑定不同的事件处理函数。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    //第 1 行代码，在文档中添加 4 个按钮
    $('<input type="button" value="第 1 个按钮"/><input type="button" value="第 2 个按钮"/><input type="button" value="删除按钮事件处理函数"/><input type="button" value="隐藏或显示文本"/>').appendTo($('body'));
    //第 2 行代码，分别选中 4 个按钮，并为它们绑定不同的事件处理函数
    $('input[type="button"]')
    .eq(0).click(function(){           //匹配第 1 个按钮，并绑定 click 事件处理函数
        alert('是第 1 个按钮的事件处理函数');
    }).end().eq(1)                    //返回所有按钮，再匹配第 2 个按钮
    .click(function(){                //为第 2 个按钮绑定 click 事件处理函数
        $('input[type="button"]:eq(0)').trigger('click');
    }).end().eq(2)                    //返回所有按钮，再匹配第 3 个按钮
    .click(function(){                //为第 3 个按钮绑定 click 事件处理函数
        $('input[type="button"]:eq(0)').unbind('click');
    }).end().eq(3)                    //返回所有按钮，再匹配第 4 个按钮
    .toggle(function(){                //为第 4 个按钮绑定 toggle 事件处理函数
        $('.panel').hide('slow');
    }, function(){
        $('.panel').show('slow');
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<div class="panel">jQuery 链式语法演示</div>
</body>
</html>
```

【代码详解】

链式代码已经成为 jQuery 非常流行的一个特点，在使用链条方式写代码时，可能会用到 eq()、filter() 等 jQuery 的方法改变链式方法的对象，但是借助 jQuery 的 end() 方法又能够恢复最初的 jQuery 对象，从而可以实现继续执行链式操作。

在上面示例中，通过 end() 方法取消当前的 jQuery 对象，返回前面的 jQuery 对象。这样当匹配某个按钮时，为其绑定事件处理函数，然后调用 end() 方法，则又返回前面一个 jQuery 对象，即按钮集合。

注意，有几个 jQuery 的方法并不返回 jQuery 对象，所以链式操作就不能继续下去，如 get() 就不能像 eq() 那样使用。

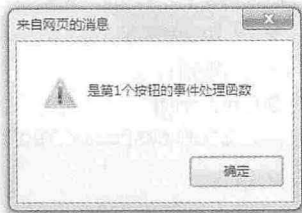


图 1.10 jQuery 链式语法应用

链式语法是一种比较时尚的编程方法，但是在使用该方法时，为了方便阅读，读者应该注意以下几个问题。

- ☑ 如果在同一个 jQuery 对象上执行不超过 3 个方法，则可以在同一行内书写。例如，下面一行代码用来选择第 1 个按钮，修改它的名称，并为其附加一个类。

```
$("input[type='button']").eq(0).val("修改按钮名称").addClass("red");
```

- ☑ 如果在同一个 jQuery 对象上执行很多操作，则应该分行进行书写，以方便阅读和修改。
- ☑ 对于多个对象执行少量的操作，则可以为每一个对象书写一行代码。如果涉及子元素操作，可以考虑使用缩进进行设计，这样就能够区分层次。例如，针对上面示例，可以这样进行缩进显示。

```
$("input[type='button']")
    .eq(0).click(function(){
        alert('you clicked me!');
    })
```

- ☑ 如果对于多个对象执行很多连续的操作，则可以考虑结合上面几种方法同时进行设计。

1.4.3 模仿 CSS 选择器让选取元素更精确、灵活

jQuery 选择器令 DOM 操作优雅而艺术，它支持 ID、tagName、CSS 1~3 表达式、XPath 表达式，详细说明请参阅 <http://docs.jquery.com/Selectors>。jQuery 不仅模仿 CSS 和 XPath 选择器的用法和功能，还自定义了很多过滤方法，综合利用这些选择器，可以随心所欲地选择 HTML 结构中的任意元素。

jQuery 选择器按照功能主要分为选择和过滤，并允许配合使用，可以同时使用组合成一个选择器字符串。两者主要的区别是：

- ☑ 过滤作用的选择器是指定条件从前面匹配的内容中筛选。过滤选择器也可以单独使用，此时表示从全部*元素中筛选。

```
$(":[title]")
```

等同于：

```
$("*:[title]")
```

- ☑ 选择功能的选择器则不会有默认的范围，因为作用是选择而不是过滤。

【示例 4】利用 jQuery 选择器快速匹配文档中的按钮，并为该按钮绑定事件处理函数。演示效果如图 1.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input[type='button']").click(function(){ //匹配提交按钮，并绑定事件处理函数
        var i = 0; //初始化临时变量
        $("input[type='text']").each(function(){ //枚举每个文本框，获取其值，然后相加
            i += parseInt($(this).val());
        });
        $("#label").text(i); //显示结果
    });
    $("input:lt(2)") //匹配非提交按钮，以及<label>标签，通过链式语法定义样式
        .add("label")
        .css("border","none")
```



```

        .css('borderBottom','solid 1px navy')
        .css('textAlign','center')
        .css('width','3em')
        .css({'width':'30px'});
    });
</script>
<title>上机练习</title>
</head>
<body>
<input type="text" value="" /> +
<input type="text" value="" />
<input type="button" value="=" />
<label></label>
</body>
</html>

```



图 1.11 jQuery 选择器应用

【代码详解】

在上面代码中，`$("input[type='button']")`选择器可以匹配文档中 `type` 属性值为 `button` 的 `input` 元素，这个表达式模仿了 CSS 表达式，然后为 `button` 添加 `click` 事件处理函数。

在 `click` 事件处理函数中，`$("input[type='text']")`选择器能够匹配文档中所有的输入框，然后调用 `each()` 方法遍历所有匹配的文本框，利用 `$(this)`选择器获取当前文本框，使用 `val()`读取当前文本框的值，再使用 JavaScript 函数 `parseInt()`把获取的字符串类型的值转换为数值类型，相加之后作为文本信息添加到 `label` 元素中显示出来。

`$('input:lt(2)')`选择器能够匹配文档中的所有 `input` 元素，然后筛选出排在前面两个的 `input` 元素，其中的伪类 `:lt` 表示序号小于某个值。匹配到 `input` 元素之后，再添加 `label` 对象，合并成一个 jQuery 对象。然后通过链式语法连续调用 3 个 `css()`方法为文本框设置样式。如果一次需要设置多个 CSS 样式，也可以使用下面的方法来进行设计。

```

.css({
    'border':'none',
    'borderBottom':'solid 1px navy',
    'width':'30px'
});

```

如果只给 `css()`方法传递一个字符串参数，则为读取样式值，如 `css('color')`就表示取得当前 jQuery 对象的样式属性 `color` 的值。而如果给它传递两个参数，则表示设置样式值。jQuery 对象定义了很多类似的方法，如 `val()`、`text()`、`html()`、`click()`、`change()`等。

注意，选择器（Selector）是一串表示特殊语义的字符，只要把选择器字符串传递给 `jQuery()`构造函数，就能够选择不同的 DOM 对象，并且返回 jQuery 对象。jQuery 选择器支持 CSS 3 选择器标准，读者可以在 W3C 官方网站（<http://www.w3c.org/TR/css3-selectors/>）了解 CSS 3 选择器的标准。

1.4.4 扩展接口让 jQuery 更开放、富有活力

jQuery 定义了庞杂的公共函数和 jQuery 方法，但是也无法满足所有人的需求。甚至可以这样说，没有哪个框架应该设法将每个人需要的东西都预先准备好。这样做可能导致一堆大而笨重的、包含很少用到的功能的代码，只会把事情搞砸。

jQuery 努力实现了扩展性，在核心库里仅定义了基础的方法和函数，但是特意留出了使得 jQuery 易于扩展的方法和接口。

【示例 5】利用 jQuery 扩展接口为 jQuery 框架定义了两个自定义函数，然后调用这两个函数。演示效果如图 1.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.fn.extend({           //调用扩展函数
    check: function() {      //为表单元素扩展 check 函数，选中单选按钮或者复选框
        return this.each(function() { this.checked = true; });
    },
    uncheck: function() {     //为表单元素扩展 check 函数，取消选中单选按钮或者复选框
        return this.each(function() { this.checked = false; });
    }
});
$(function(){               //应用扩展函数
    $("input[type=checkbox]").check().css("border","solid 1px red");
    $("input[type=radio]").uncheck().css("border","solid 1px blue");
});
</script>
<title>上机练习</title>
</head>
<body>
<input type="radio" />
<input type="radio" checked="checked" />
<input type="checkbox" />
<input type="checkbox" checked="checked" />
</body>
</html>
```

【代码详解】

在上面代码中，jQuery.fn.extend 其实是 jQuery.extend = jQuery.fn.extend = jQuery.prototype.extend。可能读者会想，如果使用 jQuery.fn.extend 增加扩展函数，在调用时应该是 \$(xxx).extend.xxxx()，而不应该是类似于上面示例代码所调用的 \$(xxx).xxxx()，当然，肯定是有代码在处理后才能这么调用。

在扩展函数 extend() 中以对象直接量的形式包含了两个自定义方法：check() 和 uncheck()。check() 方法能够为 jQuery 匹配的每个表单元素定义选中状态，uncheck() 方法能够为 jQuery 匹配的每一个表单元素定义未选中状态。

然后在页面初始化之后，使用 jQuery 构造函数匹配页面中所有的单选按钮和复选框，并设置所有单选

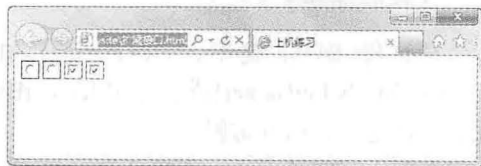


图 1.12 jQuery 扩展应用

按钮为未选中状态，而设置所有复选框为选中状态，并分别调用 `css()` 方法定义边框样式。

1.5 初学 jQuery 最容易混淆的几个概念

由于 jQuery 功能强大，提供的方法和属性众多，短期快速上手有不小的挑战。不过，这个库的设计秉承了一致性与对称性原则，它的大部分概念都是从 HTML 和 CSS 的结构中借用而来的。鉴于很多 Web 开发人员对这两种技术比对 JavaScript 更有经验，所以编程经验不多的设计者能够快速学会使用该库。

实际上，jQuery 框架本身就是在 JavaScript 语言基础上进行封装的，因此 jQuery 代码本质上也是 JavaScript 代码，自然，jQuery 代码与 JavaScript 代码可以相互混合使用。读者也没有必要去区分每一行代码到底是 jQuery 代码，还是 JavaScript 代码。但是，jQuery 与 JavaScript 是两个不同的概念，在用法上存在差异，初学者很容易被下面这些概念所迷惑，详细比较如下。

1.5.1 jQuery 对象不等于 DOM 对象

初学者很容易把 jQuery 对象和 DOM 对象混淆在一起，或者误认为两者水火不容。

DOM 是 Document Object Model 的简写，中文翻译为文档对象模型。根据 W3C DOM 规范，DOM 是 HTML 与 XML 的应用编程接口（API），DOM 将整个页面映射为一个由层次节点组成的文件。

【示例 6】 将下面文档加载并被浏览器解析后，会自动生成一个映射文件，该文件以结构树的形式展示当前文档的结构，如图 1.13 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>标准 DOM 示例</title>
  </head>
  <body>
    <h1>标准 DOM</h1>
    <p>这是一份简单的<strong>文档对象模型</strong></p>
    <ul>
      <li>D 表示文档，DOM 的物质基础</li>
      <li>O 表示对象，DOM 的思想基础</li>
      <li>M 表示模型，DOM 的方法基础</li>
    </ul>
  </body>
</html>
```

【代码详解】

在这棵 DOM 树中，``、``、`` 标签都是 `` 标签的子节点，可以通过 `document` 对象的 `getElementsByTagName()` 或者 `getElementById()` 方法访问它们，也可以利用节点之间的关系，通过它们的关联指针快速进行相互访问。

上面所有的节点和元素都是 DOM 对象的组成部分，`getElementsByTagName()` 和 `getElementById()` 方法也是 DOM 模型提供的内置方法，所有这些就构成了 DOM 对象基础。

jQuery 对象是通过 jQuery 框架包装 DOM 对象之后产生的一个新对象，本质上它是一个普通的 JavaScript 对象，该对象中包含了 DOM 对象的集合，因此可以把 DOM 对象看作是一个独立的个体，而 jQuery 可以是多个 DOM 对象组成的数据集合。

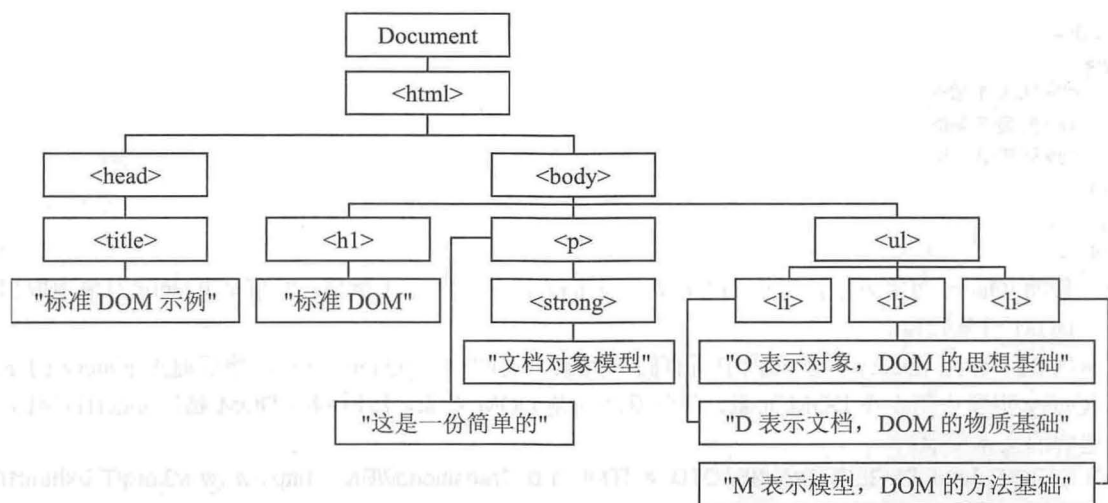


图 1.13 文档对象模型的树形结构

jQuery 框架为 jQuery 对象定义了独立使用的方法和属性，它无法直接调用 DOM 对象的方法。相反，DOM 对象也无法直接调用 jQuery 对象的方法或属性。例如，下面写法都是错误的：

```

$("#wrap").innerHTML = "嵌入文本";           //jQuery 对象调用 DOM 属性
getElementById("wrap").html("嵌入文本");       //DOM 对象调用 jQuery 对象的方法
  
```

1.5.2 jQuery 对象与 DOM 对象之间的转换

jQuery 对象和 DOM 对象是可以相互转换的，因为它们所操作的对象都是 DOM 元素，只不过 jQuery 对象包含了多个 DOM 元素，而 DOM 对象本身就是一个 DOM 元素。简单地说，jQuery 对象是 DOM 元素的数组，也称为类数组，而 DOM 对象就是一个单个 DOM 元素。

1. 把 jQuery 对象转换为 DOM 对象

jQuery 对象不能够使用 DOM 对象的方法，但是如果需要，就应该先把 jQuery 对象转换为 DOM 对象。转换的方法有以下两种。

☑ 借助数组下标来读取 jQuery 对象集合中的某个 DOM 元素对象。

【示例 7】 使用 jQuery 匹配文档中所有的 li 元素，返回一个 jQuery 对象，然后通过数组下标的方式读取 jQuery 集合中第 1 个 DOM 元素，此时返回的是 DOM 对象，然后调用 DOM 属性 innerHTML，读取该元素包含的文本信息。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var $li = $("li");           //返回 jQuery 对象
    var li = $li[0];             //返回 DOM 对象
    alert(li.innerHTML);
})
</script>
<title>上机练习</title>
</head>
  
```

```
<body>
<ul>
  <li>列表 1</li>
  <li>列表 2</li>
  <li>列表 3</li>
</ul>
</body>
</html>
```

- ☒ 借助 jQuery 对象方法，如 `get()` 方法，为 `get()` 方法传递一个下标值，即可从 jQuery 对象中取出一个 DOM 对象元素。

【示例 8】 使用 jQuery 匹配文档中所有的 `li` 元素，返回一个 jQuery 对象，然后通过 jQuery 的 `get()` 方法读取 jQuery 集合中第 1 个 DOM 元素，此时返回的是 DOM 对象，然后调用 DOM 属性 `innerHTML`，读取该元素包含的文本信息。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $li = $("li");                //返回 jQuery 对象
    var li = $li.get(0);              //返回 DOM 对象
    alert(li.innerHTML);
})
</script>
<title>上机练习</title>
</head>
<body>
<ul>
  <li>列表 1</li>
  <li>列表 2</li>
  <li>列表 3</li>
</ul>
</body>
</html>
```

2. 把 DOM 对象转换为 jQuery 对象

对于 DOM 对象来说，直接把它传递给 `$()` 函数即可，jQuery 对象会自动把它包装为 jQuery 对象，然后就可以自由调用 jQuery 定义的方法。

【示例 9】 针对上面示例，可以这样来设计，使用 DOM 的方法获取所有 `li` 元素，然后使用 `jQuery()` 构造函数把它封装为 jQuery 对象，这样就可以方便地调用 jQuery 的方法。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var li = document.getElementsByTagName("li");    //获取所有 li 元素
    var $li = $(li[0]);                             //把第 1 个 li 元素封装为 jQuery 对象
})
```



```
        alert($li.html());           //调用 jQuery 对象的方法
    })
</script>
<title>上机练习</title>
</head>
<body>
<ul>
    <li>列表 1</li>
    <li>列表 2</li>
    <li>列表 3</li>
</ul>
</body>
</html>
```

实际上,也可以把 DOM 元素数组传递给 `$()` 函数, jQuery 对象会自动把所有 DOM 元素包装在一个 jQuery 对象中。

【示例 10】 针对上面示例, 还可以进行下面的设计。

```
<script type="text/javascript" >
$(function(){
    var li = document.getElementsByTagName("li");    //获取所有 li 元素
    var $li = $(li);                                //把所有 li 元素封装为 jQuery 对象
    alert($li.html());                              //调用 jQuery 对象的方法
})
</script>
```

注意, 使用 `document.getElementsByTagName()` 方法获取的 DOM 元素集合是一个真正的数组类型, 而 jQuery 对象仅是一个类数组, 不是真正意义上的数组类型数据。

1.5.3 jQuery 的 ready 不等于 JavaScript 的 load

jQuery 定义的 `ready` 事件与 JavaScript 定义的 `load` 事件都表示页面初始化行为, 但是它们之间并非完全相同, 为了理解这两个事件的异同, 读者应该了解 HTML 文档加载的顺序。

HTML DOM 文档加载是按顺序执行的, 这与浏览器的渲染方式有关系, 一般浏览器渲染操作的顺序大致按如下几个步骤完成:

- (1) 解析 HTML 结构。
- (2) 加载外部脚本和样式表文件。
- (3) 解析并执行脚本代码。
- (4) 构造 HTML DOM 模型。
- (5) 加载图片等外部文件。
- (6) 页面加载完毕。

下面介绍二者的差异。

1. 执行时机不同

`load` 事件必须等到网页中所有内容全部加载完毕之后, 才被执行。如果一个页面中包含了大数据的多媒体文件, 则会出现网页文档已经呈现出来, 而由于网页数据还没有完全加载完毕, 导致 `load` 事件不能够即时被触发。

开发人员习惯把页面初始化设置的脚本都放在 `load` 事件处理函数中, 由于页面数据没有完全加载进来, 导致网页呈现和脚本初始化配置不能够保持同步, 从而影响了页面的可用性。

而 jQuery 的 `ready` 事件是在 DOM 结构绘制完毕之后就执行, 也就是说它先于外部文件的加载就被执行

了，这样就能够确保文档呈现和脚本初始化设置保持同步。

总之，ready 事件先于 load 事件被激活，如果网页文档中没有加载外部文件，则它们的响应时间基本上是一样的。

2. 用法不同

JavaScript 的 load 事件只能够被编写一次，下面的写法是不允许的，此时它仅能够影响最后一次指定的事件处理函数。

```
<script type="text/javascript" >
window.onload = function(){
    alert("页面初始化 1");
}
window.onload = function(){
    alert("页面初始化 2");
}
window.onload = function(){
    alert("页面初始化 3");
}
</script>
```

如果希望这 3 个事件处理函数中的代码都被执行，则应该把它们包含在一个处理函数中。

```
<script type="text/javascript" >
var f1 = function(){
    alert("页面初始化 1");
}
var f2 = function(){
    alert("页面初始化 2");
}
var f3 = function(){
    alert("页面初始化 3");
}
window.onload = function(){
    f1();
    f2();
    f3();
}
</script>
```

而对于 jQuery 的 ready 事件类型来说，可以在同一个文档中多次定义。例如，针对上面示例，可以使用 jQuery 的 ready 事件类型来设计。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    alert("页面初始化 1");
});
$(function(){
    alert("页面初始化 2");
});
$(function(){
    alert("页面初始化 3");
});
</script>
```

这对于复杂页面中多次配置初始化程序非常重要，也方便了用户根据需要随时进行设计。

1.6 学习资源

下面为读者推荐一些学习 jQuery 的相关资源, 使用这些资源, 可以帮助读者找到精通 jQuery 的捷径。

1.6.1 jQuery 开发工具

使用任何文本编辑器都可以开发 jQuery 程序, 当然使用集成的开发环境更能够提高编码效率。常用的 Web 开发工具包括 Dreamweaver、Visual Studio 等。

其中, Dreamweaver 是针对网页设计师提供的专业可视化设计工具, 从 Dreamweaver CS 5.5 版本开始, 内置了 jQuery 引擎, 实现内核支持, 并提供各种 jQuery 应用插件。

Visual Studio 是针对 .NET 开发人员提供的开发环境, 从 Visual Studio 2008 版本开发, 也内置了 jQuery 引擎, 该工具功能完善, 提供强大的环境支持, 但是占用系统资源比较大, 操作起来不是很便捷。

另外, Aptana Studio 3 版本也提供了超强的 Web 开发环境, 特别针对 JavaScript 开发, 是前端开发人员必备的工具, 该工具占用资源也比较大, 如果编写简单的代码, 不提倡使用该工具, 但是如果开发复杂的 JavaScript 脚本, 建议选用该工具。

以上工具都可以在网上搜索下载, 这里就不再提供下载链接。

1.6.2 jQuery 参考手册

学习 jQuery, 配备参考手册是必需的, 就像学汉语必须配备一本新华字典一样。首先, 读者应该在网上下载以下 jQuery 参考手册:

- ☑ jQuery 1.4 API 文档, 如图 1.14 所示, 包含 CHM 离线版。下载地址为 <http://jquery-api-zh-cn.googlecode.com/svn/trunk/xml/jqueryapi.xml>。

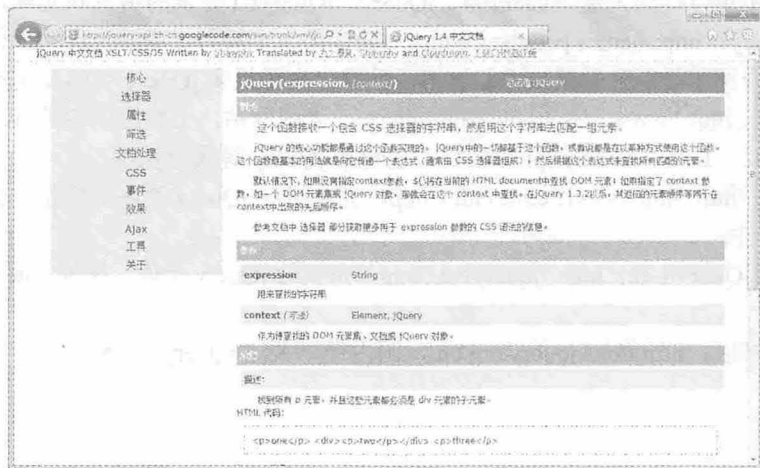


图 1.14 jQuery 1.4 API 应用文档

- ☑ jQuery 1.6 API 文档, 如图 1.15 所示。下载地址为 <http://julying.com/jquery-1.6-api/>。
- ☑ DOM 参考文档, 应该属 w3school 提供的比较完整、权威, 如图 1.16 所示。查询地址为 <http://www.w3school.com.cn/>。

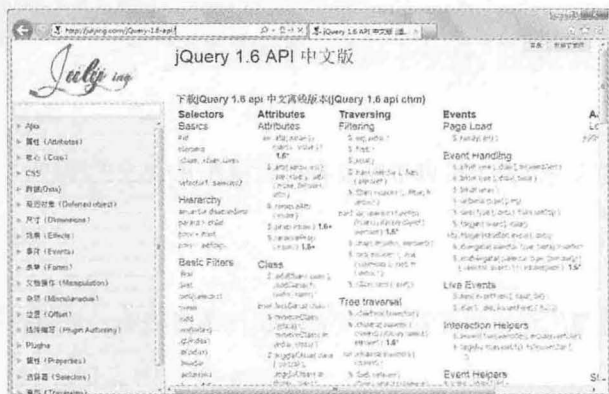


图 1.15 jQuery 1.6 API 应用文档



图 1.16 w3school 参考手册

1.6.3 jQuery 在线资源

- ☑ John Resig 博客: <http://ejohn.org/>, 毫无疑问, jQuery 作者的博客是首先必须关注的。
- ☑ Filament Group Lab: <http://www.filamentgroup.com/lab>, jQuery UI 类库出自该网站。
- ☑ Learning jQuery: <http://www.learningjquery.com/>, 老牌的 jQuery 学习网站之一。
- ☑ Soh Tnaka Blog: <http://www.sohtanaka.com/web-design-blog/>, 非常酷的 jQuery 设计和分享网站, 教程和插件实现非常有艺术感。
- ☑ nettuts: <http://net.tutsplus.com/>, 老牌的网页设计网站, 包含非常多的 jQuery 教程和技巧。同时这个网站也有大量的与设计相关的文章。
- ☑ jQuery4u: <http://www.jquery4u.com/>, 老牌的 jQuery 学习网站, 包含很多 jQuery 教程, 收集了很多 jQuery 插件。
- ☑ WebResourceDepot: <http://www.webresourcesdepot.com/>, 收集了很多 jQuery 插件和教程。
- ☑ Paul Bakus: <http://paulbaku.com/>, Paul Bakus 是 jQueryUI 的创始者, 也是很多著名插件的开发人, 在他的博客中可以找到大量 JavaScript、jQuery、jQueryUI 开发的信息和文章。
- ☑ James Padolsey: <http://james.padolsey.com/>, Jame 是一个具有丰富开发经验的前端开发工程师, 在他的博客中介绍了最新的开发技术和价格, 并且分享了很多 jQuery 代码及其实现。
- ☑ W3CSchools: <http://www.w3schools.com/jquery/>, 老牌网站, 主要提供各种 W3C 相关教程, 包括 HTML、XML、CSS、JavaScript 等教程, 也包括 jQuery 等其他教程。
- ☑ jQuery style: <http://jquerystyle.com/index.php>, 完全讲解 jQuery 类库的网站, 分享代码片段及其各种 jQuery 插件。
- ☑ The Ultimate jQuery List: <http://jquerylist.com/>, 完整的 jQuery 类库、插件、演示及其教程说明, 值得查看。
- ☑ jQuery 中文社区: <http://bbs.jquery.org.cn/>, 国内最早关注 jQuery 的网站。

第 2 章

使用选择器

( 视频讲解：1 小时 28 分钟)

jQuery 选择器是 jQuery 框架的基础，jQuery 对事件的处理、DOM 操作、CSS 动态控制、Ajax 通信、动画设计都是在选择器基础上进行的。jQuery 选择器采用 CSS 和 XPath 选择符的能力，能够满足用户在 DOM 中快捷而轻松地获取元素或元素组。本章将讲解 CSS 和 XPath 选择器，以及 jQuery 自定义的选择器。这些方法为匹配目标元素提供了更大的灵活性。

注意，在 jQuery 中通过各种选择器和方法获取的结果集合实际上都是一个 jQuery 对象。jQuery 对象有别于 DOM 对象，当读者想要实际操纵在页面中匹配的元素时，通过 jQuery 对象会非常简单，既可以轻松地 jQuery 对象绑定事件，或者添加漂亮的效果，也可以将多重修改或效果通过 jQuery 对象连缀到一起。然而，jQuery 对象与常规的 DOM 对象不同，用户不能够为 jQuery 对象调用 DOM 属性或者方法，也不能够为 DOM 对象调用 jQuery 的方法和属性。

2.1 基本选择器

基本选择器主要包括 ID 选择器、标签选择器、类选择器、通配选择器和组选择器 5 种类型，这与 CSS 基本选择器类型相一致，详细说明如表 2.1 所示。

表 2.1 jQuery 基本选择器类型

选 择 器	说 明	返 回 值
#id	根据给定的 ID 匹配一个元素。如果选择器中包含特殊字符，可以用两个斜杠转义	单个元素
element	根据指定的元素类型名称选择该类型所有元素	同类型集合元素
.class	根据指定的类名选择所有同类元素	集合元素
*	在所限定的范围内选择所有元素	所有元素的集合
selector1,selector2,selectorN	分别选择选择器组中每个选择器匹配的元素，然后合并返回所有元素	集合元素

2.1.1 ID 选择器

JavaScript 提供了原生方法 `getElementById()`，实现在 DOM 中选择指定 ID 值的元素。具体用法如下：


```
var element = document.getElementById("id");
```

其中, `getElementById()` 方法返回值为所匹配元素的对象引用, `document` 是 `Window` 对象的属性, 它引用 `Document` 对象, 参数值为字符串型 ID 值, 该值在 HTML 文档标签中通过 `id` 特性设置。

jQuery 简化了 JavaScript 原生方法的操作, 通过一个简单的 “#” 标识前缀快速匹配指定 ID 的元素对象。具体用法如下:

```
jQuery("#id");
```

参数 `id` 为字符串, 表示标签的 `id` 属性值。返回值为包含匹配 `id` 的元素的 jQuery 对象。

【示例 1】选择文档中的 ID 值为 `div1` 的元素, 并设置其背景色为红色。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#div1").css("background","red"); //页面初始化函数
    //匹配 ID 值为 div1 的元素, 并设置其背景色为红色
})
</script>
<title>上机练习</title>
</head>
<body>
<div id="div1">测试盒子</div>
</body>
</html>
```

【代码详解】

在上面代码中, `$("#div1")` 函数包含的 `#div1` 参数就表示 ID 选择器, jQuery 构造器能够根据这个选择器准确定位到 DOM 中该元素的位置, 并返回包含该元素引用的 jQuery 对象。

从本质上分析, JavaScript 与 jQuery 在选择 ID 元素时异曲同工, jQuery 只不过是包装了 `getElementById()` 方法。但是, 从执行效率来分析, 两者的差距还是很大的。由于 jQuery 需要对于参数字符串进行解析, 并匹配出所传递的参数值是 ID 值, 然后再调用 `getElementById()` 方法获取该 ID 元素, 所以所花费的时间一定会成倍地增长。因此, 在不是必需的前提下, 可以考虑直接使用 `Document` 对象的 `getElementById()` 方法获取 ID 元素。

在 ID 选择器中, 如果选择器中包含特殊字符, 可以在 jQuery 中使用两个斜杠对特殊字符进行转义。

【示例 2】页面包含 3 个 `<div>` 标签, 它们的 `id` 属性值都包含了特殊的字符。如果不进行处理, jQuery 在解析时会因为误解而不能够达到目的。此时, 可以使用如下方法来实现准确选择, 即为这些 ID 选择器字符串添加双斜杠前缀, 以便对这些特殊的字符进行转义。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#a\\.b").css("color","red");
    $("#a\\.b").css("color","red");
    $("#\\[div\\]").css("color","red");
})
```



```

</script>
<title>上机练习</title>
</head>
<body>
<div id="a.b">div1</div>
<div id="a:b">div2</div>
<div id="[div]">div3</div>
</body>
</html>

```

【代码详解】

在执行 jQuery() 函数时, jQuery 使用正则表达式来匹配参数值, 并判断当前参数是否为 ID 值:

ID: /#((?:[w\u00c0-\uffff_]\.))+/

而正则表达式对于特殊字符是敏感的, 要避免正则表达式被误解, 就考虑进行字符转义, 在正则表达式字符串中一般都通过双斜杠来转义特殊字符。

如果直接使用 JavaScript 的原生方法 getElementById() 就不用顾虑这个问题, 例如, 上面示例代码可以改写为:

```

<script type="text/javascript" >
$(function(){
    document.getElementById("a.b").style.color = "red";
    document.getElementById("a:b").style.color = "red";
    document.getElementById("[div]").style.color = "red";
})
</script>

```

2.1.2 标签选择器

JavaScript 提供了一个原生方法 getElementsByTagName(), 用来在 DOM 中选择指定类型的元素。具体用法如下:

```
var elements = document.getElementsByTagName("tagName");
```

其中, getElementsByTagName() 方法返回值为所选择类型元素集合的数组对象引用, document 是 Window 对象的属性, 它引用 Document 对象, 参数值为字符串型 HTML 标签名称。

jQuery 匹配指定标签的方法比较简单, 直接在 jQuery() 构造函数中指定标签名称即可。具体用法如下:

```
jQuery("element");
```

参数 element 为字符串, 表示标签的名称。返回值为包含匹配标签的 jQuery 对象。

但是, 与 ID 选择器不同, 标签选择器的字符串不需要附加标识前缀 (#)。

【示例 3】 使用 jQuery 构造器匹配文档中所有的 <div> 标签, 并定义它们的字体颜色为红色。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").css("color","red");
})
</script>

```

```
<title>上机练习</title>
</head>
<body>
<div>div1</div>
<div>div2</div>
<div>div3</div>
</body>
</html>
```

【代码详解】

`$("div")`构造函数表示匹配文档中所有的`<div>`标签，返回 jQuery 对象，然后调用 jQuery 的 `css()` 方法，为所有匹配的`<div>`标签定义红色字体。

如果直接使用 JavaScript 原生方法 `getElementsByName()` 匹配文档中的`<div>`标签，并设置它们的前景色为红色，则需要使用循环语句遍历返回的元素集合，并逐一设置每个元素的字体样式。实现代码如下：

```
<script type="text/javascript" >
window.onload = function(){
    var divs = document.getElementsByTagName("div");
    for(var i=0;i<divs.length;i++){
        divs[i].style.color = "red";
    }
}
</script>
```

//页面初始化函数
//返回 div 元素集合
//遍历 div 元素集合
//设置 div 元素的前景色为红色

此时`$("div")`与 `document.getElementsByTagName("div")`的运行结果一样，都返回一个元素集合对象。所以，读者还可以混合使用它们，代码如下：

```
<script type="text/javascript">
window.onload = function(){
    var divs = $("div");
    for(var i=0;i<divs.length;i++){
        divs[i].style.color = "red";
    }
}
</script>
```

//以 JavaScript 方法初始化页面处理函数
//以 jQuery 方法选择所有 div 元素
//以 JavaScript 方法遍历返回的 jQuery 结果对象

与 ID 选择器一样，jQuery 的类型选择器也存在效率低下问题。这不仅仅是因为 jQuery 需要使用正则表达式匹配选择器类型，过滤出参数值为标签名字符串，另外，由于 jQuery() 函数需要对第 1 个参数执行多路判断（即多分支条件判断），而对于标签字符串的判断位于队列的后面，不像 ID 选择器是第 1 个就被匹配判断（即第一条分支），所以这就耗费掉大量的宝贵时间。

从执行效率的角度考虑，读者应该积极考虑多使用 JavaScript 原生的 `getElementsByName()` 方法来选择同类型的元素。即使在复杂的 jQuery 编程环境中，嵌入使用 `getElementsByName()` 方法要比直接使用 `$()` 方法高效。

2.1.3 类选择器

JavaScript 没有原生的类选择方法，此时使用 jQuery 的类选择器会更加便捷。具体用法如下：

```
jQuery(".className");
```

参数 `className` 为字符串，表示标签的 `class` 属性值，前缀符号表示该选择器为类选择器。返回值为包含匹配 `className` 元素的 jQuery 对象。

【示例 4】 使用 jQuery 构造器匹配文档中所有类名为 `red` 的标签，并定义它们的字体颜色为红色。


```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $(".red").css("color","red");
})
</script>
<title>上机练习</title>
</head>
<body>
<div class="red">div1</div>
<div>div2</div>
<div class="red">div3</div>
</body>
</html>

```

在 jQuery 中，类选择器的字符串需要附加标识前缀（.）。此时 \$(".red") 返回一个包含多个元素集合的 jQuery 对象，然后直接调用 jQuery 定义的方法来操作这些匹配对象即可。

2.1.4 通配选择器

jQuery 定义了通配选择器，该选择器能够匹配指定上下文中所有元素。具体用法如下：

```
jQuery("*");
```

参数*为字符串，表示将匹配指定范围内所有的标签元素。

【示例 5】 将匹配文档中<body>标签下包含的所有标签，然后定义所有标签包含的字体显示为红色。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("body *").css("color","red");
})
</script>
<title>上机练习</title>
</head>
<body>
<div>DIV</div>
<span>SPAN</span>
<p>P</p>
</body>
</html>

```

【代码详解】

实际上，JavaScript 也支持类似的匹配方法，可以使用 `getElementsByTagName("*")` 方法快速匹配指定范围内的所有元素。例如，针对上面示例，可以使用如下的 JavaScript 原生代码控制，实现的效果是相同的：


```
<script type="text/javascript" >
$(function(){
    var all = document.getElementsByTagName("*");
    for(var i=0; i<all.length; i++){
        all[i].style.color = "red";
    }
})
</script>
```

当然，更高效的方法是把 JavaScript 原生方法和 jQuery 迭代操作相结合，这样可以提高代码执行效率，也不会多写很多代码。例如，使用 JavaScript 原生方法获取页面中所有元素，然后把这个 DOM 元素集合传递给 jQuery() 函数，把 JavaScript 数组集合封装为 jQuery 对象的类数组集合，然后借助 jQuery 的 css() 方法可以快速定义样式，从而提高整个程序的执行速度。代码如下：

```
<script type="text/javascript" >
$(function(){
    var all = document.getElementsByTagName("*");
    $(all).css("color","red");
})
</script>
```

2.1.5 组选择器

jQuery 支持 CSS 的分组选择器，通过这种方式可以扩大选择器的选择范围，同时增强 jQuery 选择器引擎的应用能力。选择多组元素可以通过逗号分隔符来分隔多个不同的选择器，这些选择器可以是任意类型的，也可以是复合选择器。具体用法如下：

```
jQuery("selector1,selector2,selectorN");
```

参数 selector1、selector2、selectorN 为字符串，表示多个选择器，这些选择器没有数量限制，它们通过逗号进行分隔。当执行组选择器之后，返回的 jQuery 对象将包含每一个选择器匹配到的元素。jQuery 在执行组选择器匹配时，先是逐一匹配每个选择器，然后将匹配到的元素合并到一个结果内并返回。

【示例 6】 利用组选择器匹配文档中包含的不同标签，然后定义所有标签包含的字体显示为红色。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("h2, #wrap, span.red, [title='text']").css("color","red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>H2</h2>
<div id="wrap">DIV</div>
<span class="red">SPAN</span>
<p title="text">P</p>
</body>
</html>
```

分组选择器的实现思路是这样的:

然后把该正则表达式的下标位置恢复到初始化位置。根据选择器字符串中的逗号作为分隔符，把选择字符串劈开，然后分别推入到 `parts` 数组中。

最后，通过条件语句分别判断 parts 数组的长度，如果长度大于 1，则重复调用 Sizzle() 函数，并分析第 1 个逗号后面的选择器字符串，依此类推。实现代码如下：

```

var Sizzle = function(selector, context, results, seed) {
    //...
    chunker.lastIndex = 0;
    //根据逗号为分隔符，劈开选择器字符串
    while ( (m = chunker.exec(selector)) !== null ) {
        parts.push( m[1] );
        if ( m[2] ) {
            extra = RegExp.rightContext;
            break;
        }
    }
    //处理多组选择器
    if ( parts.length > 1 && origPOS.exec( selector ) ) {
        //...
        //重复调用 Sizzle(), 迭代操作多组选择器的字符串
        Sizzle( parts.shift(), context );
        //...
        //处理单个选择器
    } else {
        //...
        Sizzle.find( parts.pop(), parts.length === 1 && context.parentNode ? context.parentNode : context,
isXML(context) );
        set = Sizzle.filter( ret.expr, ret.set );
        //...
    }
    //...
    return results;
};

```

2.2 层级选择器

层级选择器就是通过 DOM 嵌套关系结构来实现准确匹配, 如果想通过 DOM 元素之间的层次关系来获取特定的元素, 那么使用层级选择器将是最佳方式。层级选择器主要包括包含选择器、子选择器、相邻选择器和兄弟选择器 4 种类型, 详细说明如表 2.2 所示。

表 2.2 层级选择器

选 择 器	说 明
ancestor descendant	在给定的祖先元素下匹配所有的后代元素。ancestor 表示任何有效选择器，descendant 表示用以匹配元素的选择器，并且它是第 1 个选择器的后代元素。 例如，\$("form input")可以匹配表单下所有的 input 元素

续表

选 择 器	说 明
parent > child	在给定的父元素下匹配所有的子元素。parent 表示任何有效选择器，child 表示用以匹配元素的选择器，并且它是第 1 个选择器的子元素。 例如，\$("form > input")可以匹配表单下所有的子级 input 元素
prev + next	匹配所有紧接在 prev 元素后的 next 元素。prev 表示任何有效选择器，next 表示一个有效选择器并且紧接着第 1 个选择器。 例如，\$("label + input")可以匹配所有跟在 label 后面的 input 元素
prev ~ siblings	匹配 prev 元素之后的所有 siblings 元素。prev 表示任何有效选择器，siblings 表示一个选择器，并且它作为第 1 个选择器的同级结构。 例如，\$("form ~ input")可以匹配所有与表单同级结构的 input 元素

2.2.1 包含选择器

包含选择器就是在给定的祖先元素下匹配所有的后代元素。具体用法如下：

```
jQuery("ancestor descendant");
```

参数 ancestor 和 descendant 为字符串，其中，ancestor 表示包含选择器，descendant 表示被包含选择器。jQuery 能够在 ancestor 选择器所匹配的元素中，过滤出匹配 ancestor 选择器的所有包含元素。

【示例 7】 在文档中插入 3 个文本框，分别位于<form>标签内和外，其中第 1 个和第 2 个位于<form>标签内，而第 3 个位于<form>标签外。第 1 和第 2 个文本框分别处于不同的 DOM 层级中。

然后使用包含选择器匹配<form>标签包含的所有<input>标签，并定义被包含的文本框边框显示为红色，背景色为蓝色，预览效果如图 2.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("form input").css({"border":"solid 1px red","background":"blue"});
})
</script>
<title>上机练习</title>
</head>
<body>
<form>
<fieldset>
<label>包含的子文本框
<input />
</label>
<fieldset>
<label>包含的孙文本框
<input />
</label>
</fieldset>
</fieldset>
```



```

</form>
<label>非包含的文本框
    <input />
</label>
</body>
</html>

```

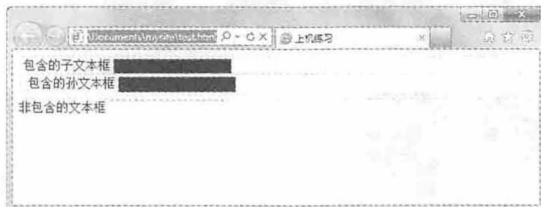


图 2.1 包含选择器的应用

注意，包含选择器不受包含结构的层级限制，只要被包含在第 1 个选择器中的所有匹配第 2 个选择器的元素都将被返回。

2.2.2 子选择器

子选择器就是在匹配的父元素下选择所有匹配的子元素。具体用法如下：

```
jQuery("parent > child");
```

参数 parent 和 child 为字符串，其中，parent 表示父选择器，child 表示被包含的子选择器，“>”为子选择器的标识符。jQuery 能够在 parent 选择器所匹配的元素中，过滤出所有匹配 child 选择器的子元素。

【示例 8】 在文档中插入 3 个图片，分别位于<div>标签内和外，其中第 1 个和第 2 个位于<div>标签内，而第 3 个位于<div>标签外。第 1、2 个文本框分别处于不同的 DOM 层级中。

然后使用子选择器匹配<div>标签包含的子标签，并定义匹配的子标签显示为红色粗边框，预览效果如图 2.2 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
img{ height:200px;}
</style>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div > img").css("border","solid 5px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <span></span>
    
</div>


```

```

</body>
</html>

```

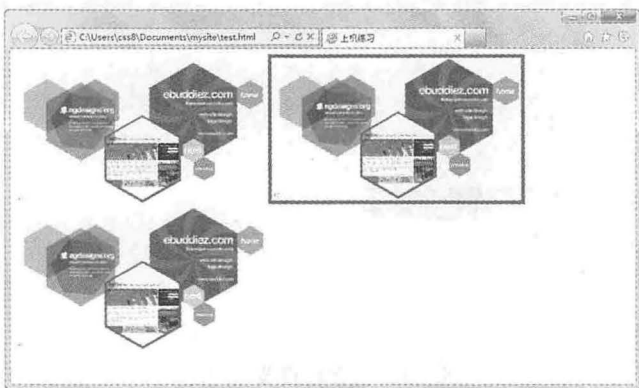


图 2.2 子选择器的应用

注意，子选择器与包含选择器在匹配结果集中有重合的部分，但是包含选择器能够匹配更多的元素，除了子元素，还包括所有嵌套的元素。

2.2.3 相邻选择器

相邻选择器就是在所有匹配的元素后选择同级的相邻元素。具体用法如下：

```
jQuery("prev + next");
```

参数 prev 和 next 为字符串，其中，prev 表示相邻的前一个选择器，next 表示相邻的后一个选择器，“+”为相邻选择器的标识符。jQuery 能够在 prev 选择器所匹配的元素后，过滤出所有匹配 next 选择器的紧邻同级元素。

【示例 9】 在文档中插入 4 个图片，分别位于<div>标签内和外，其中第 1 个和第 2 个位于<div>标签内，而第 3 个和第 4 个位于<div>标签外。第 1 个和第 2 个文本框分别处于不同的 DOM 层级中。

然后使用相邻选择器匹配<div>标签后相邻的同级标签，并定义匹配的标签显示为红色粗边框，预览效果如图 2.3 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
img{ height:200px;}
</style>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div + img").css("border","solid 5px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<div>

```

```

<span></span>

</div>


</body>
</html>

```

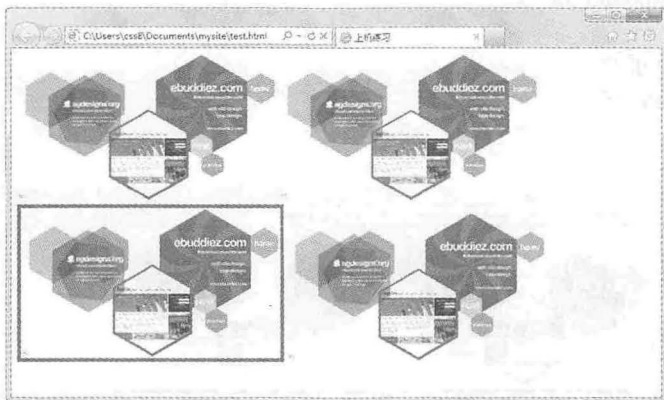


图 2.3 相邻选择器的应用

注意，与子选择器和包含选择器不同，从结构上分析相邻选择器是在同级结构上进行匹配和过滤元素，而子选择器和包含选择器是在包含的内部结构中过滤元素。

2.2.4 兄弟选择器

兄弟选择器就是在所有匹配的元素后选择同级的所有元素。具体用法如下：

```
jQuery("prev ~ siblings");
```

参数 prev 和 siblings 为字符串，其中，prev 表示相邻的前一个选择器，siblings 表示相邻的同级选择器，“~”为兄弟选择器的标识符。jQuery 能够在 prev 选择器所匹配的元素后，过滤出所有匹配 siblings 选择器的同级元素。

【示例 10】 在文档中插入 4 个图片，分别位于<div>标签内和外，其中第 1 个和第 2 个位于<div>标签内，而第 3 个和第 4 个位于<div>标签外。第 1 个和第 2 个文本框分别处于不同的 DOM 层级中。

然后使用兄弟选择器匹配<div>标签后同级的标签，并定义匹配的标签显示为红色粗边框，预览效果如图 2.4 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
img{ height:200px;}
</style>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div ~ img").css("border","solid 5px red");

```



```

})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <span></span>
    
</div>


</body>
</html>

```

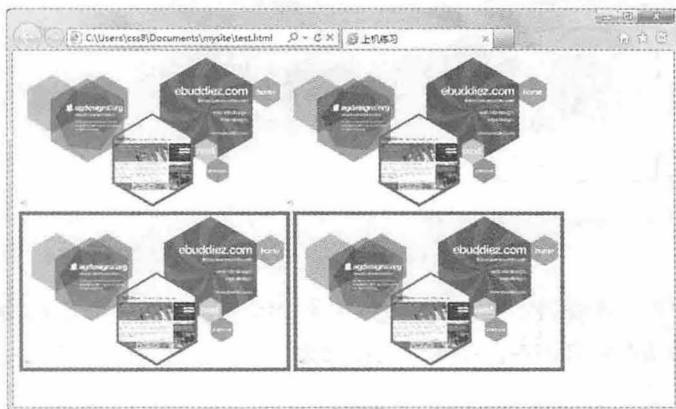


图 2.4 兄弟选择器的应用

注意，与子选择器和包含选择器不同，从结构上分析，兄弟选择器是在同级结构上进行匹配和过滤元素，而子选择器和包含选择器是在包含的内部结构中过滤元素。从这点上看，它与相邻选择器类似，但是兄弟选择器能够匹配更多的元素，除了相邻的同级元素外，还包括所有不相邻的同级元素。

2.2.5 层级选择器综合应用

当读者熟悉了层级选择器的各种形式和用法之后，就可以灵活、混合使用它们，以便在文档动态操控中随心所欲。同时在复杂的文档结构中，不需要添加任何类和 ID 属性值，因为不需要破坏文档自身结构，就可以轻松、精确匹配文档元素。

【示例 11】 利用 jQuery 定义的层级选择器可以方便地控制 HTML 文档各级元素的样式，虽然这些结构标签都没有定义 id 或 class 属性，但是这并不影响用户方便、精确地控制文档。演示效果如图 2.5 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p,div").css({"margin":"0","padding":"0.5em"}); //匹配所有的 div 和 p 元素，统一显示间距
    $("div").css("border", "solid 2px red"); //控制文档中所有 div 元素

```

```

$("div > div").css("margin", "1em"); //控制 div 元素包含的 div 子元素，实际上它与 div 包含选择器所
                                     匹配的元素是相同的
$("div div").css("background", "#ff0"); //控制最外层 div 元素包含的所有 div 元素
$("div div div").css("background", "#0f0"); //控制第 3 层及其以内的 div 元素
$("div + p").css("margin", "1em"); //控制 div 相邻的 p 元素
$("div:eq(1) ~ p").css({"background":"blue","color":"white"}); //控制 div 后面并列的所有 p 元素
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>青玉案 — 元夕</h1>
<h2>辛弃疾</h2>
<div>
  <div>东风夜放花千树，
    <div>更吹落，星如雨。</div>
    <p>宝马雕车香满路。</p>
    <p>凤箫声动，玉壶光转，</p>
    <p>一夜鱼龙舞。</p>
  </div>
  <p>蛾儿雪柳黄金缕，</p>
  <p>笑语盈盈暗香去。</p>
  <p>众里寻他千百度，</p>
</div>
<p>蓦然回首，那人却在，</p>
<p>灯火阑珊处。</p>
</body>
</html>

```

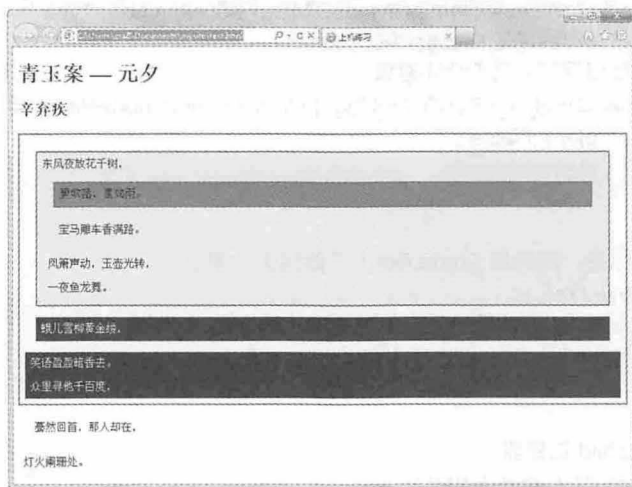


图 2.5 层级选择器的混合应用

【代码详解】

在层级选择器中，左右两个子选择器可以为任何形式的选择器，可以是基本选择器，也可以是复合选择器，甚至是层级选择器。例如，`$("div div div")`中的参数可以有两种理解：`div div`表示子包含选择器，位于左侧，作为父包含选择器的包含对象，而第 3 个 `div` 表示被包含的对象，它是一个基本选择器；或者 `div` 表示基本选择器，位于左侧，作为父包含选择器的包含对象，而 `div div` 表示被包含的对象，它是一个子包

含选择器。再如，`$("div:eq(1) ~ p")`中的 `div:eq(1)`是一个伪类选择器，它属于复合选择器，在这里表示兄弟选择器中相邻的前一个选择器。

2.2.6 解析层级选择器实现原理

jQuery 在 `Expr.relative` 对象中定义了 4 个层级选择器函数，然后在 `Sizzle()`接口函数中直接调用这些函数来匹配对应的选择器类型，并根据这些选择器表达式调用 `Sizzle.filter()`过滤函数，筛选指定关系的元素，并把这些匹配元素封装到 jQuery 对象中返回。代码如下：

```
var Expr = Sizzle.selectors = {
  relative: {
    //匹配 prev + next 选择器
    "+": function(checkSet, part, isXML){
      //处理标签字符串
      var isPartStr = typeof part === "string",
          isTag = isPartStr && !/\W/.test(part),
          isPartStrNotTag = isPartStr && !isTag;
      //把标签转换为大写形式
      if ( isTag && !isXML ) {
        part = part.toUpperCase();
      }
      //遍历检测结果集
      for ( var i = 0, l = checkSet.length, elem; i < l; i++ ) {
        //获得 elem 的前一个节点
        if ( (elem = checkSet[i]) ) {
          //返回已选元素的上一个同属节点（同级节点中的上一个）
          //当 elem 的节点类型不为元素节点时
          //继续得到 elem 的前一个节点，否则循环结束
          while ( (elem = elem.previousSibling) && elem.nodeType !== 1 ) {}
          //筛选符合指定标签的相邻元素
          //处理返回同级 DOM 对象
          checkSet[i] = isPartStrNotTag || elem && elem.nodeName === part ?
            elem || false :
            elem === part;
        }
      }
      //如果不是标签，则调用 Sizzle.filter()函数筛选结果集
      if ( isPartStrNotTag ) {
        //在 checkSet 中以后的元素中过滤选择器 part
        Sizzle.filter( part, checkSet, true );
      }
    },
    //匹配 parent > child 选择器
    ">": function(checkSet, part, isXML){
      //part 是选择器
      var isPartStr = typeof part === "string";
      //当 part 为单词字符时，如$( "form > input" )，part 为 form
      if ( isPartStr && !/\W/.test(part) ) {
        part = isXML ? part : part.toUpperCase();
        //遍历检测结果集
        for ( var i = 0, l = checkSet.length; i < l; i++ ) {
          var elem = checkSet[i];
```



```

        if ( elem ) {
            //获得 elem 的父节点
            var parent = elem.parentNode;
            //如果父节点名称为 part 值时, 在 checkSet[i]上赋值父节点, 否则赋值 false
            checkSet[i] = parent.nodeName === part ? parent : false;
        }
    }
    //当 part 为非单词字符时, 如$(".red > input"), part 为.red
} else {
    //遍历检测结果集
    for ( var i = 0, l = checkSet.length; i < l; i++ ) {
        var elem = checkSet[i];
        if ( elem ) {
            checkSet[i] = isPartStr ?
                elem.parentNode :
                elem.parentNode === part;
        }
    }
    //如果不是标签, 则调用 Sizzle.filter()函数筛选结果集
    if ( isPartStr ) {
        Sizzle.filter( part, checkSet, true );
    }
}
},
//匹配 ancestor descendant 选择器
"": function(checkSet, part, isXML){
    var doneName = done++, checkFn = dirCheck;
    if ( !part.match(/^\W/) ) {
        var nodeCheck = part = isXML ? part : part.toUpperCase();
        checkFn = dirNodeCheck;
    }
    checkFn("parentNode", part, doneName, checkSet, nodeCheck, isXML);
},
//匹配 prev ~ siblings 选择器
"~": function(checkSet, part, isXML){
    var doneName = done++, checkFn = dirCheck;
    if ( typeof part === "string" && !part.match(/^\W/) ) {
        var nodeCheck = part = isXML ? part : part.toUpperCase();
        checkFn = dirNodeCheck;
    }
    checkFn("previousSibling", part, doneName, checkSet, nodeCheck, isXML);
}
}
}
}

```

在匹配 ancestor descendant 选择器和 prev~siblings 选择器时, 用到了 dirNodeCheck()和 dirCheck()两个内部函数, 代码如下:

//检测节点型操作符函数

```

function dirNodeCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
    var sibDir = dir == "previousSibling" && !isXML;
    //遍历结果集
    for ( var i = 0, l = checkSet.length; i < l; i++ ) {
        var elem = checkSet[i];
    }
}

```

```

if ( elem ) {    //如果结果集中存在该节点
    //如果节点为元素，且操作符为 previousSibling
    if ( sibDir && elem.nodeType === 1 ){
        elem.sizcache = doneName;
        elem.sizset = i;
    }
    elem = elem[dir];
    var match = false;
    while ( elem ) {
        if ( elem.sizcache === doneName ) {
            match = checkSet[elem.sizset];
            break;
        }
        if ( elem.nodeType === 1 && !isXML ){
            elem.sizcache = doneName;
            elem.sizset = i;
        }
        if ( elem.nodeName === cur ) {
            match = elem;
            break;
        }
        elem = elem[dir];
    }
    checkSet[i] = match;
}
}

//检测操作符函数
function dirCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
    var sibDir = dir == "previousSibling" && !isXML;
    for ( var i = 0, l = checkSet.length; i < l; i++ ) {
        var elem = checkSet[i];
        if ( elem ) {
            if ( sibDir && elem.nodeType === 1 ) {
                elem.sizcache = doneName;
                elem.sizset = i;
            }
            elem = elem[dir];
            var match = false;
            while ( elem ) {
                if ( elem.sizcache === doneName ) {
                    match = checkSet[elem.sizset];
                    break;
                }
                if ( elem.nodeType === 1 ) {
                    if ( !isXML ) {
                        elem.sizcache = doneName;
                        elem.sizset = i;
                    }
                }
                if ( typeof cur !== "string" ) {
                    if ( elem === cur ) {
                        match = true;
                    }
                }
            }
        }
    }
}

```

```
        break;
    }
    } else if ( Sizzle.filter( cur, [elem] ).length > 0 ) {
        match = elem;
        break;
    }
}
//由于这里 dir 为 previousSibling, 所以这里利用循环不断得到 elem 的前一个节点,
//并且赋值 checkSet 数组
elem = elem[dir];
}
checkSet[i] = match;
}
}
```

2.3 简单的伪类选择器

伪类可以看作是一种特殊的类选择符, 是能被浏览器自动识别的特殊选择符。伪类选择器的最大语法特征就是在选择符中添加“:”标识符。与 CSS 伪类选择器相比, jQuery 定义了庞大的伪类选择器, 这些选择器能够根据用户的不同需求, 提出不同的匹配服务。

由于 jQuery 定义了大量的伪类选择器, 所以本书也将分类进行讲解。本节将介绍一些简单的伪类选择器, 主要说明如表 2.3 所示。

表 2.3 简单的伪类选择器

选 择 器	说 明
:first	匹配找到的第 1 个元素。例如, <code>\$("tr:first")</code> 表示匹配表格的第 1 行
:last	匹配找到的最后一个元素。例如, <code>\$("tr:last")</code> 表示匹配表格的最后一行
:not	去除所有与给定选择器匹配的元素。注意, 在 jQuery 1.3 中, 已经支持了复杂选择器, 如 <code>:not(div a)</code> 和 <code>:not(div,a)</code> 。例如, <code>\$("input:not(:checked)")</code> 可以匹配所有未选中的 input 元素
:even	匹配所有索引值为偶数的元素, 从 0 开始计数。例如, <code>\$("tr:even")</code> 可以匹配表格的 1、3、5 行 (即索引值 0、2、4...)
:odd	匹配所有索引值为奇数的元素, 从 0 开始计数。例如, <code>\$("tr:odd")</code> 可以匹配表格的 2、4、6 行 (即索引值 1、3、5...)
:eq	匹配一个给定索引值的元素, 从 0 开始计数。例如, <code>\$("tr:eq(0)")</code> 可以匹配第 1 行表格行
:gt	匹配所有大于给定索引值的元素, 从 0 开始计数。例如, <code>\$("tr:gt(0)")</code> 可以匹配第 2 行及其后面行
:lt	匹配所有小于给定索引值的元素。例如, <code>\$("tr:lt(1)")</code> 可以匹配第 1 行及其后面行
:header	匹配如 h1、h2、h3 之类的标题元素
:animated	匹配所有正在执行动画效果的元素

注意, 简单伪类选择器也被称为定位过滤器, 因为它们主要根据编号和排位筛选特定位置上的元素, 或者过滤掉特定元素。

2.3.1 特定位置选择器

特定位置选择器主要包括 `:first`、`:last` 和 `:eq(index)` 3 种。`:first` 能够在匹配的集合中选择第 1 个元素, 相反 `:last`

能够在匹配的集合中选择最后一个元素，而:eq(index)能够根据 index 索引值确定指定位置的元素。具体用法如下：

```
jQuery("selector:first");
jQuery("selector:last");
jQuery("selector:eq(index)");
```

参数 selector、eq(index)为字符串，其中，selector 表示任意形式的选择器，“:”为伪类选择器的标识符。

eq(index)虽然以字符串形式进行传递，但是 eq()自身却是一个匹配函数，该函数接收一个 index 索引值，这个索引值从 0 开始计数，与数组下标位置一一对应。

【示例 12】在下面文档中插入了一个 3 行 2 列的数据表格。使用特定定位选择器，分别匹配数据表第 1 行和最后一行，然后分别为它们定义不同的显示样式，同时使用 eq(1)匹配函数获取数据表中第 2 行对象，并定义它的背景样式。预览效果如图 2.6 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("tr:first").css({"background":"blue","color":"white"});    //匹配第 1 行表格
    $("tr:eq(1)").css("background", "#F1F5FB");                //匹配第 2 行表格
    $("tr:last").css("background", "#ff9");                    //匹配第 3 行表格
})
</script>
<title>上机练习</title>
</head>
<body>
<table>
<tr><th>选择器</th><th>说明</th></tr>
<tr><td>:first</td><td>匹配找到的第 1 个元素。例如，$("tr:first")表示匹配表格的第 1 行</td></tr>
<tr><td>:last</td><td>匹配找到的最后一个元素。例如，$("tr:last")表示匹配表格的最后一行</td></tr>
</table>
</body>
</html>
```

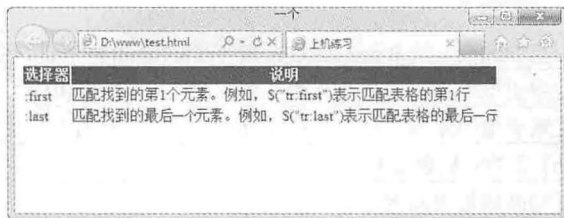


图 2.6 特定位置选择器的应用

注意，特定位置选择器是针对伪类分隔符前面的选择器所匹配的结果基础进行第 2 次过滤的，如果没有指定匹配的范围，则将视为是整个文档范围。例如，针对上面示例，如果按如下方法进行匹配：

```
<script type="text/javascript" >
$(function(){
    $("tr:first").css({"background":"blue","color":"white"});
    $("tr:eq(1)").css("background", "#F1F5FB");
```

```

$(":last").css("background", "#ff9");
})
</script>

```

则\$(":first")将匹配到文档中的第1个元素,即<html>标签,而\$(":last")将匹配到文档中的最后一个元素,即<td>标签,\$(":eq(1)")将匹配到文档中第2个元素,即<head>标签,此时预览效果如图2.7所示。

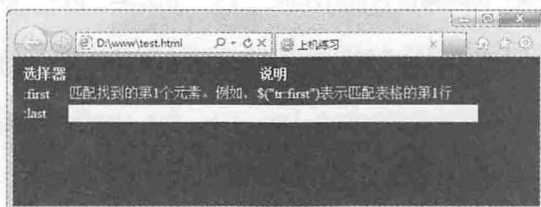


图 2.7 特定位置选择器的应用

2.3.2 指定范围选择器

特定范围选择器主要包括:even、:odd、:gt(index)和:lt(index)4种。:even能够在匹配的集合中选择所有偶数行元素,相反,:odd能够在匹配的集合中选择所有奇数行元素,:gt(index)能够在匹配的集合中选择大于给定索引值的元素,:lt(index)能够在匹配的集合中选择小于给定索引值的元素。具体用法如下:

```

jQuery("selector:even");
jQuery("selector:odd");
jQuery("selector:gt(index)");
jQuery("selector:lt(index)");

```

参数 selector、even 和 odd 为字符串,其中,selector 表示任意形式的选择器,even 表示在匹配元素中选择偶数行元素,odd 表示在匹配元素中选择奇数行元素,匹配元素集合第1个元素下标从0开始计数。“:”为伪类选择器的标识符。index 表示数字索引值,从0开始。

【示例 13】在下面文档中插入了一个8行2列的数据表格。使用指定范围选择器,分别匹配数据表的偶数行和奇数行,然后分别为它们定义不同的显示样式,最后再使用\$(":tr:first")选择器匹配第1行表格,为它单独设计标题行样式。预览效果如图2.8所示。

```

<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(":tr:even").css("background", "#F1F5FB");
    $(":tr:odd").css("background", "#ff9");
    $(":tr:first").css({"background":"blue","color":"white"});
})
</script>
<title>上机练习</title>
</head>
<body>
<table>
<tr><th>选择器</th><th>说明</th></tr>
<tr><td>:first</td><td>匹配找到的第1个元素。例如,$("&quot;tr:first&quot;")表示匹配表格的第1行</td></tr>
<tr><td>:last</td><td>匹配找到的最后一个元素。例如,$("&quot;tr:last&quot;")表示匹配表格的最后一行

```

```

</td></tr>
<tr><td>:even</td><td>匹配所有索引值为偶数的元素，从 0 开始计数。例如，$("tr:even")可以匹
配表格的 1、3、5 行（即索引值 0、2、4...） </td></tr>
<tr><td>:odd</td><td>匹配所有索引值为奇数的元素，从 0 开始计数。例如，$("tr:odd")可以匹配
表格的 2、4、6 行（即索引值 1、3、5...） </td></tr>
<tr><td>:eq</td><td>匹配一个给定索引值的元素，从 0 开始计数。例如，$("tr:eq(0)")可以匹配第
1 行表格行 </td></tr>
<tr><td>:gt</td><td>匹配所有大于给定索引值的元素，从 0 开始计数。例如，$("tr:gt(0)")可以匹
配第 2 行及其后面行 </td></tr>
<tr><td>:lt</td><td>匹配所有小于给定索引值的元素。例如，$("tr:lt(1)")可以匹配第 1 行及其后面行
</td></tr>
</table>
</body>
</html>

```

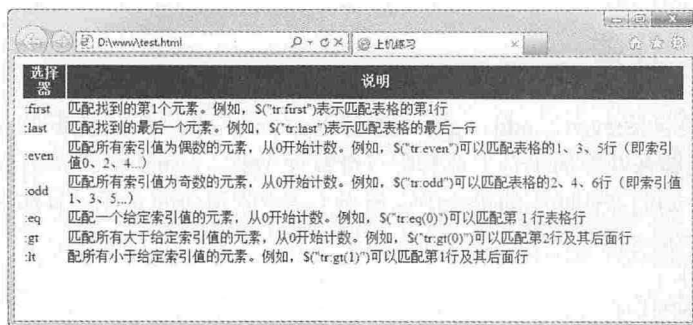


图 2.8 指定范围选择器的应用

注意，:gt(index)和:lt(index)选择器能够匹配连续的多个元素，而:even 和:odd 仅能够匹配非连续的多个元素。这些选择器在表格样式和列表样式设计中应用价值比较大。

2.3.3 排除选择器

:not 选择器比较特殊，它能够在匹配元素集合中排除符合特定匹配规则的元素，也就是说它以反向方式快速过滤掉不需要的元素。具体用法如下：

```
jQuery("selector1:not(selector2)");
```

参数 selector1、selector2 和 not 为字符串，其中，selector1 和 selector2 表示任意形式的选择器，":" 为伪类选择器的标识符，not 表示排除函数标识符。

【示例 14】在下面文档中插入了一个 8 行 2 列的数据表格。使用指定范围选择器，分别匹配数据表偶数行和奇数行，然后分别为它们定义不同的显示样式，最后再使用("tr:not(tr:lt(3))")选择器恢复第 4 行到数据表末尾所有行的默认样式。预览效果如图 2.9 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("tr:even").css("background", "#F1F5FB");
    $("tr:odd").css("background", "#fff9");

```



```

$("tr:first").css({"background":"blue","color":"white"});
$("tr:not(tr:lt(3))").css({"background":"white","color":"black"});
})
</script>
<title>上机练习</title>
</head>
<body>
<table>
<tr><th>选择器</th><th>说明</th></tr>
<tr><td>:first</td><td>匹配找到的第 1 个元素。例如, $("tr:first")表示匹配表格的第 1 行 </td></tr>
<tr><td>:last</td><td>匹配找到的最后一个元素。例如, $("tr:last")表示匹配表格的最后一行
</td></tr>
<tr><td>:even</td><td>匹配所有索引值为偶数的元素, 从 0 开始计数。例如, $("tr:even")可以匹
配表格的 1、3、5 行 (即索引值 0、2、4...) </td></tr>
<tr><td>:odd</td><td>匹配所有索引值为奇数的元素, 从 0 开始计数。例如, $("tr:odd")可以匹配
表格的 2、4、6 行 (即索引值 1、3、5...) </td></tr>
<tr><td>:eq</td><td>匹配一个给定索引值的元素, 从 0 开始计数。例如, $("tr:eq(0)")可以匹配第
1 行表格行 </td></tr>
<tr><td>:gt</td><td>匹配所有大于给定索引值的元素, 从 0 开始计数。例如, $("tr:gt(0)")可以匹
配第 2 行及其后面行 </td></tr>
<tr><td>:lt</td><td>匹配所有小于给定索引值的元素。例如, $("tr:lt(1)")可以匹配第 1 行及其后面行
</td></tr>
</table>
</body>
</html>

```

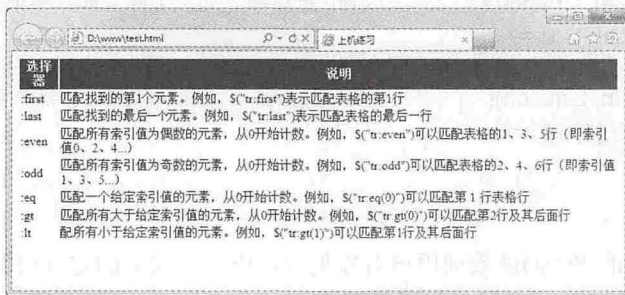


图 2.9 排除选择器的应用

【代码详解】

在\$("tr:not(tr:lt(3))")复合选择器中, tr:lt(3)将匹配数据表中第 1、2、3 行, 然后使用排除函数 not()把这些匹配的行过滤掉, 则最后匹配的数据行为第 4 行和后面的所有行, 并为这些行设计默认的背景色和前景色样式。

2.3.4 特殊选择器

:animated 和:header 选择器是两个比较特殊的选择器, 它们分别用来匹配动画元素和标题元素。具体用法如下:

```

jQuery("selector: animated");
jQuery("selector: header");

```

参数 selector、animated 和 header 为字符串。其中, selector 表示任意形式的选择器, ":" 为伪类选择器的标识符, animated 表示正在执行动画效果的元素, header 表示标题元素, 如 h1、h2、h3 等。

2.3.5 解析简单伪类选择器的实现原理

jQuery 在 Expr.setFilters 对象中收集了各种定位过滤器的表达式算法。代码如下：

```
var Expr = Sizzle.selectors = {
  setFilters: {
    first: function(elem, i){           //:first 选择器，如果是第 1 个元素，则返回 true
      return i === 0;
    },
    last: function(elem, i, match, array){ //:last 选择器，如果是最后一个元素，则返回 true
      return i === array.length - 1;
    },
    even: function(elem, i){           //:even 选择器，如果下标值是 2 的倍数，则返回 true
      return i % 2 === 0;
    },
    odd: function(elem, i){            //:odd 选择器，如果下标值不是 2 的倍数，则返回 true
      return i % 2 === 1;
    },
    lt: function(elem, i, match){       //:lt 选择器，如果下标值小于 0，则返回 true
      return i < match[3] - 0;
    },
    gt: function(elem, i, match){       //:gt 选择器，如果下标值大于 0，则返回 true
      return i > match[3] - 0;
    },
    nth: function(elem, i, match){      //:nth 选择器，如果下标值等于某个值，则返回 true
      return match[3] - 0 == i;
    },
    eq: function(elem, i, match){       //:eq 选择器，如果下标值等于某个值，则返回 true
      return match[3] - 0 == i;
    }
  }
}
```

然后在 Expr.filter 对象的 POS()函数调用该对象集合，根据所设置的定位过滤器表达式，调用 filter()函数匹配对应的元素，并返回 jQuery 对象。代码如下：

```
var Expr = Sizzle.selectors = {
  filter: {
    POS: function(elem, match, i, array){
      var name = match[2], filter = Expr.setFilters[ name ];
      if ( filter ) {
        return filter( elem, i, match, array );
      }
    }
  }
}
```

2.4 与内容相关的伪类选择器

内容伪类选择器主要是根据元素包含内容作为筛选条件进行匹配，这类选择器主要包括 4 种，主要说

明如表 2.4 所示。

表 2.4 内容伪类选择器

选 择 器	说 明
:contains	匹配包含给定文本的元素。例如, \$("div:contains('图片')")匹配所有包含“图片”的 div 元素
:empty	匹配所有不包含子元素或者文本的空元素
:has	匹配含有选择器所匹配的元素元素。例如, \$("div:has(p)")匹配所有包含 p 元素的 div 元素
:parent	匹配含有子元素或者文本的元素

2.4.1 匹配包含文本选择器

:contains(text)选择器是一个比较实用的针对网页文本过滤的选择器,它能够根据指定的文本在所能匹配的元素集合中搜索包含特定关键字的元素。具体用法如下:

```
jQuery("selector:contains(text)");
```

参数 selector、contains、text 为字符串。其中, selector 表示任意形式的选择器,“:”为伪类选择器的标识符, contains()表示文本匹配函数, text 表示一个用以查找的字符串。

【示例 15】 在下面文档的评论列表中, 首先匹配所有<dd>标签, 然后过滤<dd>标签包含“乔布斯”关键字的评论, 并为该条评论设计一条下划线。预览效果如图 2.10 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("dd:contains('乔布斯')").css("text-decoration", "underline");
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>纪念乔布斯的评论:</h2>
<dl>
<dt>李德 0202:</dt>
<dd>.....中国有这样的人那该多好呢, 苹果公司的总现金竟然超过了美国政府, 成了世界最赚钱的公司, 真希望他并未死, 为世界的人民创造更多的奇迹!!!!!! </dd>
<dt>我要杀你的那个:</dt>
<dd>乔布斯走了, 他留给我们的是无限的美好, 他是一位伟人, 不可超越的伟人, 但是他又这样走了。他留给我们的是未来! 史蒂夫·乔布斯! 1955-2011! 永远铭记在我们心里! 乔布斯, 一路走好! </dd>
<dt>阳光_太刺眼:</dt>
<dd>天才, 佩服</dd>
</dl>
</body>
</html>
```

注意, 在 contains()过滤函数中必须使用单引号包含文本关键字, 否则 jQuery 将无法进行识别和解析。

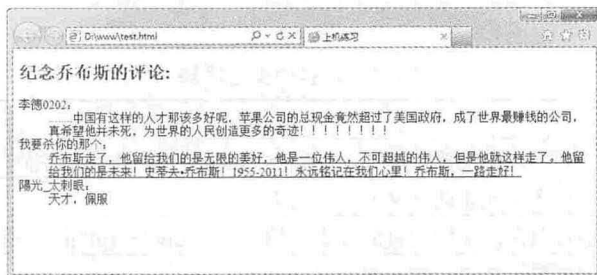


图 2.10 匹配包含文本选择器的应用

2.4.2 匹配包含元素选择器

:has(selector)选择器与:contains(text)选择器用法类似, 但是:has(selector)主要搜索匹配元素所包含的元素进行过滤。具体用法如下:

```
jQuery("selector1:has(selector2)");
```

参数 selector1、selector2、has 为字符串。其中, selector1 和 selector2 表示任意形式的选择器, ":" 为伪类选择器的标识符, has()表示包含元素匹配函数。

【示例 16】 在文档中插入 4 个导航图标, 通过子选择器匹配所有<a>标签包含的图像不显示边框样式, 然后使用:has(selector)选择器过滤出文档中所有包含标签的<a>标签, 并为它们定义一个边框样式效果。预览效果如图 2.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("a>img").css("border", "none");
    $("a:has(img)").css("border", "solid 2px #ddd");
})
</script>
<title>上机练习</title>
</head>
<body>
<a href="#"></a>
<a href="#"></a>
<a href="#"></a>
<a href="#"></a>
</body>
</html>
```

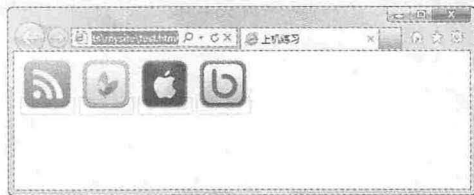


图 2.11 匹配包含元素选择器的应用

注意, `:has(selector)` 选择器是对冒号前面的选择器所匹配的元素进行二次过滤, 而不是选择 `has(selector)` 函数中 `selector` 选择器所匹配的元素。

2.4.3 包含判断选择器

`:empty` 和 `:parent` 选择器比较特殊, 它们专门用来检测匹配元素是否包含内容。其中, `:empty` 选择器过滤出匹配元素中包含空内容的元素, 而 `:parent` 选择器过滤出匹配元素中包含非空内容的元素。具体用法如下:

```
jQuery("selector: empty");
jQuery("selector: parent");
```

参数 `selector`、`empty`、`parent` 为字符串。其中, `selector` 表示任意形式的选择器, “.” 为伪类选择器的标识符, `empty` 表示包含空内容的匹配元素, `parent` 表示包含内容的匹配元素。

2.4.4 解析内容过滤器实现原理

jQuery 在 `Expr.filters` 对象中收集了各种内容过滤器的表达式算法, 代码如下:

```
var Expr = Sizzle.selectors = {
  filters: {
    parent: function(elem) {           //:parent 选择器, 如果存在第 1 个子元素, 则返回 true
      return !!elem.firstChild;
    },
    empty: function(elem) {           //:empty 选择器, 如果不存在第 1 个子元素, 则返回 true
      return !elem.firstChild;
    },
    has: function(elem, i, match) {    //:has 选择器, 调用 Sizzle() 函数检测指定的表达式所匹配的元素是否存在, 如果存在, 则返回 true
      return !!Sizzle( match[3], elem ).length;
    }
  }
}
```

其中当 `!elem.firstChild` (即 `elem` 元素) 不包含子节点或者文本元素时, `empty` 返回真; 当 `!!elem.firstChild` (即 `elem` 元素) 包含子节点或者文本元素时, `parent` 返回真; 在 `:has` 选择器中, `match[3]` 为 `has` 紧跟在后面含有的元素, 如 `$("div:has(p)")` 中的 `p`, `!!Sizzle(match[3], elem).length` 获得 `match[3]` 元素中包含在 `elem` 元素中的个数, 如果个数大于 1, 则 `has` 返回真。

然后在 `Expr.filter` 对象的 `PSEUDO()` 函数中调用该对象集合, 根据所设置的定位过滤器表达式, 调用 `filter()` 函数匹配对应的元素, 并返回 jQuery 对象。代码如下:

```
var Expr = Sizzle.selectors = {
  filter: {
    PSEUDO: function(elem, match, i, array) {
      var name = match[1], filter = Expr.filters[ name ];
      if ( filter ) {
        //匹配:parent、:empty 和:has 选择器
        return filter( elem, i, match, array );
      } else if ( name === "contains" ) {
        //匹配:contains 选择器
        // textContext 在 FF 下和 innerText 在 IE 下的属性是等效的
        // match[3]得到的是 contains 紧跟在后面包含的字符串
        //当 elem 元素的文本内容包含 contains 包含的关键字时, 返回 true
        return (elem.textContent || elem.innerText || "").indexOf(match[3]) >= 0;
      } else if ( name === "not" ) {
        var not = match[3];
```

{

当返回真时, elem 元素为匹配的元素。

可见或有隐藏进行快速过滤，详细说明如表 2.5 所示。

1990		1991		1992		1993		1994		1995		1996		1997		1998		1999		2000		2001		2002		2003		2004		2005		2006		2007		2008		2009		2010		2011		2012		2013		2014		2015		2016		2017		2018		2019		2020		2021		2022		2023		2024		2025		2026		2027		2028		2029		2030		2031		2032		2033		2034		2035		2036		2037		2038		2039		2040		2041		2042		2043		2044		2045		2046		2047		2048		2049		2050		2051		2052		2053		2054		2055		2056		2057		2058		2059		2060		2061		2062		2063		2064		2065		2066		2067		2068		2069		2070		2071		2072		2073		2074		2075		2076		2077		2078		2079		2080		2081		2082		2083		2084		2085		2086		2087		2088		2089		2090		2091		2092		2093		2094		2095		2096		2097		2098		2099		2100	
1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100																																																																																																															

.visible	匹配所有的可见元素
----------	-----------

过 p.madden 过滤器匹配它们，并把它们显示出来。预览效果如图 2.12 所示。

1

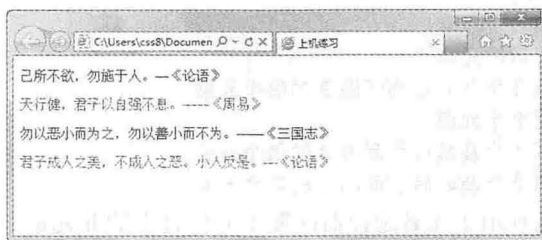


图 2.12 显隐伪类选择器的应用

【原理解析】

jQuery 专门为:hidden、:visible 和:animated 3 个伪选择器定制了 3 个独立的公共函数，然后在 filter() 中调用这些函数来匹配特殊的过滤器。具体代码如下：

```
Sizzle.selectors.filters.hidden = function(elem){
    return elem.offsetWidth === 0 || elem.offsetHeight === 0; //匹配:hidden 选择器
};
Sizzle.selectors.filters.visible = function(elem){
    return elem.offsetWidth > 0 || elem.offsetHeight > 0; //匹配:visible 选择器
};
Sizzle.selectors.filters.animated = function(elem){ //匹配:animated 选择器
    return jQuery.grep(jQuery.timers, function(fn){
        return elem === fn.elem;
    }).length;
};
```

当 elem 元素的 CSS 属性 offsetWidth 为 0，或者 offsetHeight 为 0 时，hidden 返回真；当 elem 元素的 CSS 属性 offsetWidth 不为 0，或者 offsetHeight 不为 0 时，visible 返回真。jQuery 通过布尔值来判断元素是否显示。

2.6 匹配子元素的伪类选择器

与内容相关的伪类选择器能够根据元素所包含的内容来过滤元素，而与子元素相关的伪类选择器却能够根据所匹配的元素检索其包含的子元素，这与层级选择器中的子选择器有几分相似。简单来说，子元素选择器就是通过当前匹配元素选择该元素包含的特定子元素。子元素选择器主要包括 4 种类型，简单说明如表 2.6 所示。

表 2.6 子元素伪类选择器

选 择 器	说 明
:nth-child	匹配其父元素下的第 N 个子或奇偶元素
:first-child	匹配第一个子元素。 :first 选择器只匹配一个元素，而:first-child 选择符将为每个父元素匹配一个子元素
:last-child	匹配最后一个子元素。 :last 只匹配一个元素，而:last-child 选择符将为每个父元素匹配一个子元素
:only-child	如果某个元素是父元素中唯一的子元素，那将会被匹配；如果父元素中含有其他元素，那将不会被匹配

注意，后面将介绍到:eq(index)选择器，该选择器只能够匹配一个元素，而:nth-child 能够为每一个父元素匹配子元素。:nth-child 是从 1 开始的，而:eq()是从 0 算起的。下面表达式都是可以使用的：

【示例 18】下面示例分别利用子元素选择器匹配不同位置上的 li 元素，并为其设计不同的样式，演示如图 2.13 所示。

己所不欲，勿施于人。——《论语》

天行健，君子以自强不息。——《周易》

勿以恶小而为之，勿以善小而不为。——《三国志》

君子成人之美，不成人之恶。小人反是。——《论语》

图 2.13 子元素选择器的应用

【原理解析】

```
var Expr = Sizzle.selectors = {
  order: [ "ID", "NAME", "TAG" ],
  match: {
    ID: /#((?:[\w\u00c0-\uFFFF_]|\\.)+)/,
    CLASS: /\.((?:[\w\u00c0-\uFFFF_]|\\.)+)/,
    NAME: /\[name=[""]*((?:[\w\u00c0-\uFFFF_]|\\.)+)[""]*\]/,
    ATTR: /\[s*(((?:[\w\u00c0-\uFFFF_]|\\.)+)\s*(?:\s*(?:\s*=\s*))\s*((?:[""]*)\s*(.*?)\s*))\s*\]/,
```

```

TAG: /^(?:[\w\u00c0-\uFFFF_-\]|\\.)+/,
CHILD: /:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^~]|$)/,
PSEUDO: /:(?:(?:[\w\u00c0-\uFFFF_-\]|\\.)+)(?:\((["']*)(?:\((?:[\^\2\\])+\)+)2\)))/
}
}

```

然后，在 `Sizzle.filter()` 过滤器函数中调用该正则表达式匹配到子元素选择器特征字符，并调用 `Expr.preFilter` 对象中包含的 `CHILD()` 方法。最后把处理所得的匹配元素封装到 jQuery 对象中返回，代码如下：

```

if ( Expr.preFilter[ type ] ) {
    match = Expr.preFilter[ type ]( match, curLoop, inplace, result, not, isXMLFilter );
    if ( !match ) {
        anyFound = found = true;
    } else if ( match === true ) {
        continue;
    }
}
}

```

`CHILD()` 方法位于 `Expr.preFilter` 对象中，详细代码如下：

```

var Expr = Sizzle.selectors = {
    preFilter: {
        CHILD: function(match) {
            if ( match[1] == "nth" ) {
                // parse equations like 'even', 'odd', '5', '2n', '3n+2', '4n-1', '-n+6'
                var test = /(-?)(\d*)n((?:\+|-)?\d*)/.exec(
                    match[2] == "even" && "2n" || match[2] == "odd" && "2n+1" ||
                    !/\D/.test( match[2] ) && "0n+" + match[2] || match[2]);
                // calculate the numbers (first)n+(last) including if they are negative
                match[2] = (test[1] + (test[2] || 1)) - 0;
                match[3] = test[3] - 0;
            }
            // TODO: Move to normal caching system
            match[0] = done++;
            return match;
        },
    },
}

```

2.7 与表单对象相关的伪类选择器

表单对象比较多，在网页设计中使用频率也比较高，但是很多表单对象都使用 `input` 元素来定义。为了方便用户灵活匹配表单对象，jQuery 定义了很多与表单对象相关的伪类选择器，简单说明如表 2.7 所示。使用这些选择器可以方便地获取表单中的各种表单对象。

表 2.7 表单对象选择器

选 择 器	说 明
<code>:input</code>	匹配所有 <code>input</code> 、 <code>textarea</code> 、 <code>select</code> 和 <code>button</code> 元素
<code>:text</code>	匹配所有的单行文本框
<code>:password</code>	匹配所有密码框
<code>:radio</code>	匹配所有单选按钮

续表

选 择 器	说 明
:checkbox	匹配所有复选框
:submit	匹配所有提交按钮
:image	匹配所有图像域
:reset	匹配所有重置按钮
:button	匹配所有按钮
:file	匹配所有文件域
:hidden	匹配所有不可见元素, 或者 type 为 hidden 的元素

【示例 19】 下面示例演示了如何使用表单对象选择器快速匹配不同类型的表单对象, 并分别定义它们的显示文本和样式。演示效果如图 2.14 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("#test :text").val("文本框");
    $("#test :password").val("密码域");
    $("#test :checkbox").val("复选框").css("border","solid 1px red");
    $("#test :radio").val("单选按钮").css("border","solid 1px blue");
    $("#test :image").val("图像域").css("border","solid 1px green");
    $("#test :file").val("文件域").css("border","solid 1px pink");
    $("#test :hidden").val("隐藏域");
    $("#test :button").val("普通按钮");
    $("#test :submit").val("提交按钮");
    $("#test :reset").val("重置按钮");
})
</script>
<title>上机练习</title>
</head>
<body>
<form id="test" action="" method="get">
    <input name="" type="text" value=""><br />
    <input name="" type="password" value=""><br />
    <input name="" type="checkbox" value="复选框"><br />
    <input name="" type="radio" value="单选按钮"><br />
    <input name="" type="image" value="" src="images/btn.gif" S><br />
    <input name="" type="file" value=""><br />
    <input name="" type="hidden" value=""><br />
    <input name="" type="button" value=""><br />
    <input name="" type="submit" value=""><br />
    <input name="" type="reset" value=""><br />
</form>
</body>
</html>
```

第2章 使用选择器

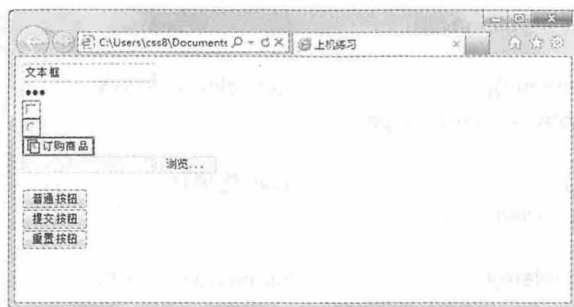


图 2.14 表单对象选择器的应用

【原理解析】

jQuery 在 Sizzle.selectors 表达式对象中定义了 filters 子对象，该子对象是一个表达式过滤函数集，主要负责过滤表单域中的各种特征域。如果匹配的元素符合该特征，则返回 true，否则返回 false。详细代码如下。

首先，jQuery 将调用下面正则表达式匹配到表单选择器字符：

PSEUDO: /:(?:(?:[w\u00c0-\uFFFF_-]|\\.)+)(?:\\([\"']*(?:\\.|\\(?!\\)|\\s)*\\)|\\[(^\\|\\s)*\\])+\\2\\)?/

并通过匹配到的选择器特征字符找到它们的核心代码：

```
var Expr = Sizzle.selectors = {
  filters: {
    enabled: function(elem){           //:enabled 选择器
      return elem.disabled === false && elem.type !== "hidden";
    },
    disabled: function(elem){          //:disabled 选择器
      return elem.disabled === true;
    },
    checked: function(elem){           //:checked 选择器
      return elem.checked === true;
    },
    selected: function(elem){          //:selected 选择器
      // 访问该属性，选择在默认情况下启用
      // 在 Safari 中选项正常工作
      elem.parentNode.selectedIndex;
      return elem.selected === true;
    },
    parent: function(elem){            //:parent 选择器
      return !elem.firstChild;
    },
    empty: function(elem){             //:empty 选择器
      return !elem.firstChild;
    },
    has: function(elem, i, match){      //:has 选择器
      return !!Sizzle( match[3], elem ).length;
    },
    header: function(elem){            //:header 选择器
      return /h\d/i.test( elem.nodeName );
    },
    text: function(elem){               //:text 选择器
      return "text" === elem.type;
    },
    radio: function(elem){              //:radio 选择器
```



```

        return "radio" === elem.type;
    },
    checkbox: function(elem){                //:checkbox 选择器
        return "checkbox" === elem.type;
    },
    file: function(elem){                    //:file 选择器
        return "file" === elem.type;
    },
    password: function(elem){                //:password 选择器
        return "password" === elem.type;
    },
    submit: function(elem){                  //:submit 选择器
        return "submit" === elem.type;
    },
    image: function(elem){                  //:image 选择器
        return "image" === elem.type;
    },
    reset: function(elem){                  //:reset 选择器
        return "reset" === elem.type;
    },
    button: function(elem){                  //:button 选择器
        return "button" === elem.type || elem.nodeName.toUpperCase() === "BUTTON";
    },
    input: function(elem){                  //:input 选择器
        return /input|select|textarea|button/i.test(elem.nodeName);
    }
}
}

```

elem.type 获得元素的 type 属性，当元素 type 属性等于相应的值时，返回相应的布尔值。如果为真，最后返回匹配的 jQuery 对象。其中，enabled 对应 elem 的 disabled 属性为 false，并且 elem 的 type 属性为 hidden；disabled 对应 elem 的 disabled 属性为 true；checked 对应 elem 的 checked 属性为 true；selected 对应 elem 的 selected 属性为 true。

然后，jQuery 在 Sizzle.selectors 表达式对象中定义了 filter 子对象，该子对象是一个选择器类型过滤函数集，主要负责过滤选择器表达式的类型。如果匹配的选择器类型是伪类选择器，则将调用该对象集合包含的 PSEUDO() 方法。PSEUDO() 方法首先调用 Expr.filters 集合对象，查询选择器的名称，并根据选择器的名称确定所要执行的操作。代码如下：

```

var Expr = Sizzle.selectors = {
    filter: {
        PSEUDO: function(elem, match, i, array){
            var name = match[1], filter = Expr.filters[ name ];
            //如果是表单类型选择器，则返回调用该表单类型选择器对应的值
            //Expr.filters[ name ] ()函数的值，返回 true 或者 false
            //最后 Sizzle 选择器和过滤器再根据这个返回值，逐一筛选匹配的表单域元素
            if ( filter ) {
                return filter( elem, i, match, array );
            } else if ( name === "contains" ) {
                return (elem.textContent || elem.innerText || "").indexOf(match[3]) >= 0;
            } else if ( name === "not" ) {
                var not = match[3];
                for ( var i = 0, l = not.length; i < l; i++ ) {

```



```
        if ( not[i] === elem ) {  
            return false;  
        }  
    }  
    return true;  
}  
}  
}
```

2.8 与表单属性相关的伪类选择器

除了表单对象选择器外, jQuery 还根据表单域中特有的属性定义了 4 个表单属性选择器, 这些选择器与表单对象选择器不同, 通过它们可以选择任何类型的表单域, 因为它主要根据表单属性来进行选择, 说明如表 2.8 所示。

表 2.8 表单属性选择器

选 择 器	说 明
:enabled	匹配所有可用元素
:disabled	匹配所有不可用元素
:checked	匹配所有选中的被选中元素 (复选框、单选按钮等, 不包括 select 中的 option)
:selected	匹配所有选中的 option 元素

【示例 20】 下面示例演示了如何使用表单对象选择器快速匹配不同类型的表单对象, 并分别定义它们的显示文本和样式。演示效果如图 2.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/  
xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml"><head>  
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />  
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>  
<script type="text/javascript" >  
$(function(){  
    $("#test :disabled").val("不可用");  
    $("#test :enabled").val("可用");  
    $("#test :checked").removeAttr("checked");  
    $("#test :selected").removeAttr("selected");  
})  
</script>  
<title>上机练习</title>  
</head>  
<body>  
<form id="test" action="" method="get">  
    <input name="" type="text" disabled="disabled" value="文本域"><br />  
    <input name="" type="text" disabled="disabled" value="文本域"><br />  
    <input name="" type="text" value="文本域"><br />  
    <input name="" type="checkbox" checked="checked" value="复选框">复选框<br />  
    <input name="" type="radio" value="单选按钮">单选按钮<br />  
    <select name="">
```

```

<option value="1">1</option>
<option value="1">2</option>
<option value="1" selected="selected">3</option>
</select>
</form>
</body>
</html>

```

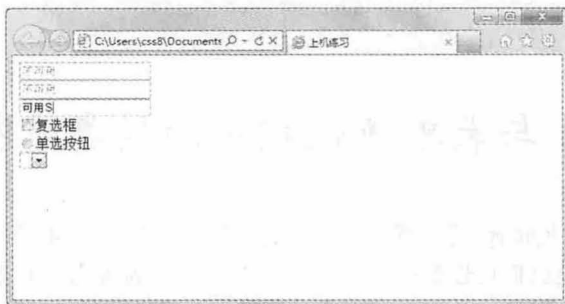


图 2.15 表单属性选择器的应用

2.9 属性选择器

属性选择器就是根据元素的属性及其值作为过滤条件，来匹配对应的 DOM 元素。属性选择器一般都是以中括号作为起止分界符的，如`[attribute]`，它与伪类选择器特征都比较明显。jQuery 定义了 7 类属性选择器，说明如表 2.9 所示。

表 2.9 属性选择器

选 择 器	说 明
<code>[attribute]</code>	匹配包含给定属性的元素。 注意，在 jQuery 1.3 中，前导的“@”符号已经被废除，如果想要兼容最新版本，只需要简单去掉“@”符号即可。例如， <code>\$(“div[id]”)</code> 表示查找所有含有 id 属性的 div 元素
<code>[attribute=value]</code>	匹配属性等于特定值的元素。属性值的引号在大多数情况下是可选的，如果属性值中包含诸“]”时，需要加引号用以避免冲突。 例如， <code>\$(“input[name='text']”)</code> 表示查找所有 name 属性值是 text 的 input 元素
<code>[attribute!=value]</code>	匹配所有不含有指定的属性，或者属性不等于特定值的元素。该选择器等价于 <code>:not([attr=value])</code> 。 要匹配含有特定属性但不等于特定值的元素，可以使用 <code>[attr]:not([attr=value])</code> 。 例如， <code>\$(“input[name!='text']”)</code> 表示查找所有 name 属性值不是 text 的 input 元素
<code>[attribute^=value]</code>	匹配给定的属性是以某些值开始的元素。 例如， <code>\$(“input[name^='text']”)</code> 表示所有 name 属性值以 text 开始的 input 元素
<code>[attribute\$=value]</code>	匹配给定的属性是以某些值结尾的元素。 例如， <code>\$(“input[name\$='text']”)</code> 表示所有 name 属性值以 text 结束的 input 元素
<code>[attribute*=value]</code>	匹配给定的属性是以包含某些值的元素。 例如， <code>\$(“input[name*='text']”)</code> 表示所有 name 属性值包含 text 字符串的 input 元素
<code>[selector1][selector2][selectorN]</code>	复合属性选择器，需要同时满足多个条件时使用。 例如， <code>\$(“input[name*='text'] [id]”)</code> 表示所有 name 属性值包含 text 字符串，且包含了 id 属性的 input 元素

匹配属性名选择器是一种简单的属性选择器，它能够为包含指定属性名的所有该类型标签定义样式。

【示例 21】 下面示例演示了如何使用匹配属性名选择器快速匹配设置了 alt 属性的标签，并分别定义它们的 CSS 样式。演示效果如图 2.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("img").css("width","260px");
    $("img[alt]").css("border","solid 2px red");
})
</script>
<title>上机练习</title>
</head>
<body>



</body>
</html>
```

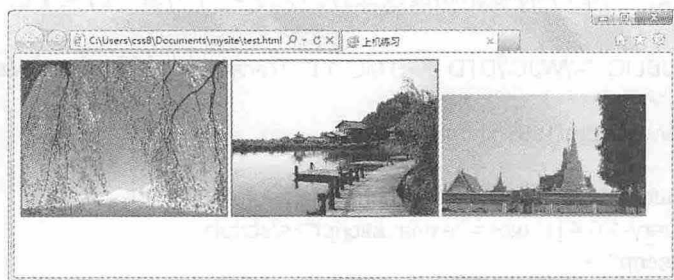


图 2.16 匹配属性名选择器的应用

匹配属性值选择器是指根据元素的属性值进行匹配，在指定属性值时应该确保值被单引号括起来。

【示例 22】 下面示例演示了如何使用匹配属性值选择器快速匹配设置了 alt 属性，且设置 title 属性值为“图像”的标签，并分别定义它们的 CSS 样式。演示效果如图 2.17 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("img").css("width","260px");
    $("img[alt][title='图像']").css("border","solid 2px red");
})
</script>
<title>上机练习</title>
</head>
<body>

```



```


</body>
</html>
```

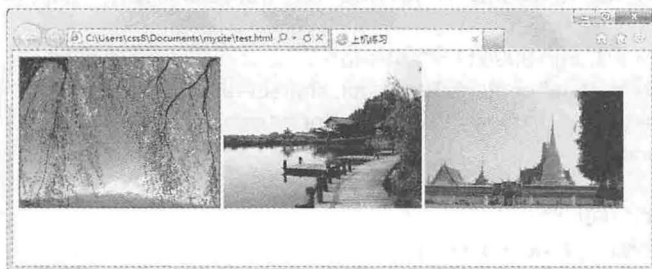


图 2.17 匹配属性值选择器的应用

模糊匹配属性值选择器是一类特殊的属性选择器，类似于正则表达式的匹配模式，也是属性选择器中功能最强大的一部分功能。它主要包括如下几种匹配模式。

- ☑ `[!=]`（非匹配）：匹配不等于特定属性值。
- ☑ `[^=]`（前缀匹配）：匹配属性值中的起始字符。
- ☑ `[$=]`（后缀匹配）：匹配属性值中的结束字符。
- ☑ `[*=]`（子字符串匹配）：匹配属性值存在的指定字符。

【示例 23】 在下面这个示例中将根据超链接文件的类型，分别为不同类型的文件添加类型文件图标。演示效果如图 2.18 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("a[href$='.pdf']").before(" <img src='images/pdf.gif' /> ");
    $("a[href$='.rar']").before(" <img src='images/rar.gif' /> ");
    $("a[href$='.jpg'],a[href$='.bmp'],a[href$='.gif'],a[href$='.png']").before(" <img src='images/jpg.gif' /> ");
    $("a[href^='http:']").before(" <img src='images/html.gif' /> ");
});
</script>
<title>上机练习</title>
</head>
<body>
<ul>
<li><a href="1.pdf">1.pdf</a></li>
<li><a href="2.rar">2.rar</a></li>
<li><a href="3.jpg">3.jpg</a></li>
<li><a href="4.bmp">4.bmp</a></li>
<li><a href="5.gif">5.gif</a></li>
<li><a href="6.png">6.png</a></li>
<li><a href="http://www.baidu.com/">http://www.baidu.com/</a></li>
</ul>
</body>
</html>
```

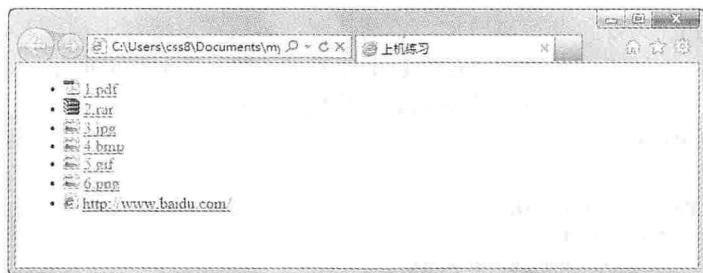


图 2.18 属性选择器综合应用

【代码详解】

在上面示例中，首先使用属性选择器匹配标签的 href 属性值结尾是否包含相应的扩展名，然后根据扩展名为该标签添加前缀图标，即通过 before() 方法为超链接标签补加对应类型的图标。

【原理解析】

属性选择器的实现相对比较复杂。首先，jQuery 在选择器中调用 Expr.match[ATTR] 集合对象匹配属性选择器字符串的基本特征，以中括号字符为起止分界符。代码如下：

```
var Expr = Sizzle.selectors = {
  order: [ "ID", "NAME", "TAG" ],
  match: {
    ID: /#((?:[\w\u00c0-\uFFFF_]|\\.)+)/,
    CLASS: /\.(?:[\w\u00c0-\uFFFF_]|\\.)+/,
    NAME: /\[name=['"]*((?:[\w\u00c0-\uFFFF_]|\\.)+)[/*"]]/,
    ATTR: /\[s*((?:[\w\u00c0-\uFFFF_]|\\.)+)\s*(?:\S?=\s*(['"]*)(.*)\s*)\]/,
    TAG: /^((?:[\w\u00c0-\uFFFF_*]|\\.)+)/,
    CHILD: /\:(only|nth|last|first)-child(?:\(\s*(even|odd|(dn|+-)*)\s*\))?/,
    POS: /\:(nth|eq|gt|lt|first|last|even|odd)\s*(?:\(\s*(\d*)\s*\))?(?:\s*=\s*(['"]*)(.*)\s*)?/,
    PSEUDO: /\:(?:[\w\u00c0-\uFFFF_]|\\.)+((?:\s*(['"]*)(.*)\s*)\s*)?/,
  },
};
```

其中属性匹配的正则表达式为 ATTR: /\[s*((?:[\w\u00c0-\uFFFF_]|\\.)+)\s*(?:\S?=\s*(['"]*)(.*)\s*)\]/。

然后，通过 Expr.preFilter[ATTR] 函数预处理属性选择器字符串中的特殊格式，并返回正则表达式匹配结果的数组。代码如下：

```
var Expr = Sizzle.selectors = {
  preFilter: {
    ATTR: function(match, curLoop, inplace, result, not, isXML){
      var name = match[1].replace(/\\/g, "");
      //映射 class 和 for 属性
      if ( !isXML && Expr.attrMap[name] ) {
        match[1] = Expr.attrMap[name];
      }
      //为兄弟选择器添加空格
      if ( match[2] === "~=" ) {
        match[4] = " " + match[4] + " ";
      }
      //返回匹配的数组
      return match;
    }
  }
};
```

最后，通过 `Sizzle.filter` 方法，获得 `ATTR` 的正则匹配，然后在过滤器中调用 `Expr.filter[ATTR]` 函数匹配属性选择器字符串的基本特征，以中括号字符为起止分界符。如果匹配对应的字符串，则返回 `true`，即选择当前元素，否则返回 `false`，表示放弃选择当前元素。代码如下：

```
var Expr = Sizzle.selectors = {
  filter: {
    ATTR: function( elem, match ) {
      var name = match[1],
          result = Expr.attrHandle[ name ] ?
            Expr.attrHandle[ name ]( elem ) :
            elem[ name ] != null ?
              elem[ name ] :
              elem.getAttribute( name ),
          value = result + "",
          type = match[2],
          check = match[4];

      // 返回匹配结果，以布尔值表示，true 表示选择，false 表示不选择
      return result == null ?
        type === "!=" :
        type === "=" ?
          value === check :
          type === "*=" ?
            value.indexOf( check ) >= 0 :
            type === "~=" ?
              ( " " + value + " ").indexOf( check ) >= 0 :
              !check ?
                value && result !== false :
                type === "!=" ?
                  value !== check :
                  type === "^=" ?
                    value.indexOf( check ) === 0 :
                    type === "$=" ?
                      value.substr( value.length - check.length ) === check :
                      type === "|=" ?
                        value === check || value.substr( 0, check.length + 1 ) === check + "-":
                        false;
    }
  }
}
```

其中 `value` 相当于 `newsletter`，`check` 相当于 `new`，则各种属性选择器的说明如下：

- ☑ “!=” 和 “=” 判断 `value === check` 的布尔值，即 `value` 是否等于 `check`。
- ☑ “^=” 取得 `value.index(check) === 0` 的布尔值，即 `check` 的字符串是否在 `value` 的开头。
- ☑ “\$=” 取得 `value.substr(value.length - check.length) === check` 的布尔值，即 `check` 的字符串是否在 `value` 的末尾。
- ☑ “*=” 取得 `value.index(check) >= 0` 的布尔值，即 `value` 包含 `check` 字符串为真。

2.10 jQuery 选择器应用优化

正确使用选择器引擎对于页面性能起了至关重要的作用。使用合适的选择器表达式可以提高性能、增

强语义并简化逻辑。在传统用法中，最常用的简单选择器包括 ID 选择器、Class 选择器、类型标签选择器，其中 ID 选择器是速度最快的。这主要是因为 JavaScript 内置函数 `getElementById()`。其次是类型选择器，因为使用 JavaScript 内置函数 `getElementsByTag()`，速度最慢的是 Class 选择器，其需要通过解析 HTML 文档树，并且需要在浏览器内核外递归，这种递归遍历是无法被优化的。

就需求分析，CSS 的选择器是为了通过语义来渲染样式，而 jQuery 的选择器只是为了选出一类 `DOMElement`，执行逻辑操作。但是，在实际开发中，Class 选择器是使用频率最高的类型之一，如表 2.10 所示。

表 2.10 jQuery 选择器使用频率列表

选 择 器	统 计 频 率
#id	51.290%
.class	13.082%
tag	6.416%
tag.class	3.978%
#id tag	18.151%
tag#id	1.935%
#id:visible	1.577%
#id .class	1.434%
.class .class	1.183%
*	0.968%
#id tag.class	0.932%
#id:hidden	0.789%
tag[name=value]	0.645%
.class tag	0.573%
[name=value]	0.538%
tag tag	0.502%
#id #id	0.430%
#id tag tag	0.358%

Class 选择器在文档中使用频率靠前，这无疑会增加系统的负担，因为每一次使用 Class 选择器，整个文档就会被解析一遍，并遍历每个节点。因此，建议读者在使用 jQuery 选择器时，应该注意以下几个问题：

(1) 多用 ID 选择器。

多用 ID 选择器，这是一个明智的选择，如果不存在 ID 选择器，则可以从父级元素中添加一个 ID 选择器，这样就会缩短节点访问的路程。

(2) 少直接使用 Class 选择器。

可以使用复合选择器，例如，使用 `tag.class` 代替 `.class`。文档的标签是有限的，但是类可以拓展标签的语义，那么大部分情况下，使用同一个类的标签也是相同的。

当然，应该摒除表达式中的冗余部分，对于不必要的复合表达式就应该进行简化，例如，`#id2 #id1` 或者 `tag#id1` 表达式，不妨直接使用 `#id1` 即可，因为 ID 选择器是唯一的，执行速度最快。使用复合选择器，则会增加负担。

(3) 多用父子关系，少用嵌套关系。

例如，使用 `parent>child` 代替 `parent child`。因为“>”是 child 选择器，只从子节点里匹配，不递归。而“ ”是后代选择器，递归匹配所有子节点及子节点的子节点，即后代节点。

(4) 缓存 jQuery 对象。

如果选出结果不发生变化的话，不妨缓存 jQuery 对象，这样就可以提高系统性能，养成缓存 jQuery 对象的习惯，可以让用户在不经意间就能够完成主要的性能优化。

例如，下面的用法是低效的：


```
for (i = 0 ; i < 100 ; i ++ ) ... {  
    var myList = $( '.myList' );  
    myList.append(i);  
}
```

而使用下面方法先缓存 jQuery 对象，则执行效率就会大大提高：

```
var myList = $( '.myList' );  
for (i = 0 ; i < 100 ; i ++ ) ... {  
    myList.append(i);  
}
```

第 3 章

使用过滤器

( 视频讲解：55 分钟)

jQuery 提供了两种选择文档元素的方式：选择器和过滤器。选择器主要模仿 CSS 和 XPath 语法，提供高效、准确匹配元素的一般方法和用法，而过滤器是建立在选择器基础上进行的二次筛选。选择器是符合一定规律的字符串组合，而过滤器就是一系列简单、实用的 jQuery 方法。在 jQuery 框架中，过滤器通过 Sizzle.filter 子类来实现，它包含过滤、查找和串联 3 类操作行为，本章将结合具体的示例详细讲解这些工具的应用。

3.1 过 滤

过滤是现有的 jQuery 对象所包含元素进行再筛选的操作，jQuery 过滤方法主要包括 9 种，详细说明如表 3.1 所示。

表 3.1 jQuery 过滤方法

过 滤 方 法	说 明
eq(index)	获取第 N 个元素
hasClass(class)	检查当前的元素是否含有某个特定的类，如果有，则返回 true
filter(expr)	筛选出与指定表达式匹配的元素集合
filter(fn)	筛选出与指定函数返回值匹配的元素集合
is(expr)	用一个表达式来检查当前选择的元素集合，如果其中至少有一个元素符合这个给定的表达式就返回 true
map(callback)	将一组元素转换成其他数组（不论是否是元素数组）
has(expr)	保留包含特定后代的元素，去掉那些不含有指定后代的元素
not(expr)	删除与指定表达式匹配的元素
slice(start,[end])	选取一个匹配的子集

3.1.1 类过滤

类过滤就是根据元素的类属性来进行过滤操作，jQuery 通过 hasClass() 方法来实现。具体用法如下：
hasClass(className)

参数 `className` 是一个字符串，表示元素的类名。该方法适合执行判断操作，判断当前 jQuery 对象中的某个元素是否包含了指定类名，如果包含则返回 `true`，否则返回 `false`。但是该方法无法过滤包含特定类名的元素。

【示例 1】 在回调函数中适合使用 `hasClass()` 方法来对 jQuery 对象中包含的每个元素进行类型过滤，并根据其是否包含特定类型来执行指定行为。在下面示例中，设置当 `<div>` 标签包含 `class` 属性值为 `red` 的元素时，则为其绑定一组动画，实现当鼠标单击类名为 `red` 的 `<div>` 标签时，让它左右摆动两下。演示效果如图 3.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").click(function(){           //为所有 div 元素绑定单击事件
        if ( $(this).hasClass("red") )   //只有类名为 red 的 div 元素才绑定系列动画
            $(this)
                .animate({ left: 120 })
                .animate({ left: 240 })
                .animate({ left: 0 })
                .animate({ left: 240 })
                .animate({ left: 120 });
    });
})
</script>
<style type="text/css">
div{
    position:absolute;
    width:100px;
    height:100px;
}
.blue { background:blue; left:0px;}
.red { background:red; left:120px; z-index:2;}
.green { background:green; left:240px;}
.pos { top:120px;}
</style>
<title>上机练习</title>
</head>
<body>
<div class="blue"></div>
<div class="red"> </div>
<div class="green"></div>
<div class="red pos"> </div>
</body>
</html>
```

【代码详解】

在上面代码中，文档中共包含了 4 个 `<div>` 标签，其中有两个 `<div>` 标签包含了 `red` 类名，在包含 `red` 类名的 `<div>` 标签中，有一个是复合类名，即不仅包含 `red` 类，还包含了 `pos` 类。在页面初始化构造函数中，

使用 `jQuery()` 函数匹配文档中所有的 `div` 元素，然后为它们绑定 `click` 事件。

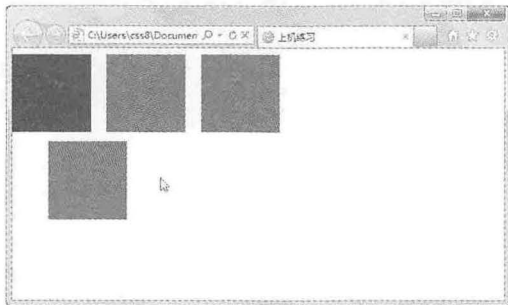


图 3.1 过滤指定类元素并为其绑定动画

在事件处理函数中检测每个元素是否包含 `red` 类。如果包含，则为其绑定系列动画，实现当用户单击红色盒子时，它能够左右摇摆显示。

3.1.2 下标过滤

类过滤还仅是一个条件检测，无法真正过滤出符合指定类名的元素，但是使用下标过滤就可以精确选出 `jQuery` 对象中指定下标的元素。`eq()` 方法的用法如下：

`eq(index)`

参数 `index` 是一个整数值，从 0 开始，用来指定元素在 `jQuery` 对象中的下标位置。

【示例 2】 针对示例 1，下面借助 `eq()` 方法精确选取出第 2 个 `<div>` 标签，并为其绑定一组动画，此时第 4 个 `<div>` 标签（即第 2 个红色盒子）就不会拥有该动画行为。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").eq(1).click(function() { //为第 2 个 div 元素绑定系列动画
        $(this)
            .animate({ left: 120 })
            .animate({ left: 240 })
            .animate({ left: 0 })
            .animate({ left: 240 })
            .animate({ left: 120 });
    });
});
</script>
<style type="text/css">
div{
    position: absolute;
    width: 100px;
    height: 100px;
}
.blue { background: blue; left: 0px; }
.red { background: red; left: 120px; z-index: 2; }
```

```
.green { background:green; left:240px;}
.pos { top:120px;}
</style>
<title>上机练习</title>
</head>
<body>
<div class="blue"></div>
<div class="red"> </div>
<div class="green"></div>
<div class="red pos"> </div>
</body>
</html>
```

3.1.3 表达式过滤

表达式过滤是最强大的过滤工具，因为表达式具有较大的灵活性，用户可以根据需要自定义表达式的形式，只要表达式符合 jQuery 选择器语法形式即可，可以是简单的选择器表达式，也可以是复杂的复合选择器表达式。

1. filter()方法

filter()方法是功能最强大的表达式过滤器，同时也可以接收函数参数，并根据函数的返回值来确定要过滤的元素。具体用法如下：

```
filter(expr)
filter(fn)
```

参数 expr 表示 jQuery 选择器表达式字符串，fn 表示函数。

【示例 3】 在下面示例中使用 filter()方法从\$("div")所匹配的 div 元素集合中过滤出包含 red 类的元素，然后为这些元素定义红色背景。演示效果如图 3.2 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").filter(".red").css("background-color","red");
})
</script>
<style type="text/css">
div{ height:20px;}
</style>
<title>上机练习</title>
</head>
<body>
<div class="blue"></div>
<div class="red"> </div>
<div class="green"></div>
<div class="red pos"> </div>
</body>
</html>
```

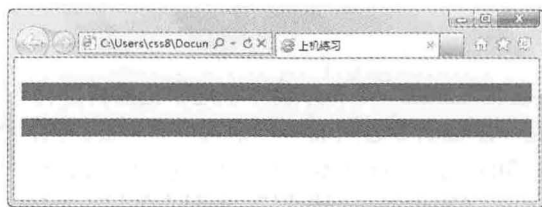



图 3.2 使用 filter()方法过滤元素

【提示】

该方法还可以带多个表达式，表达式之间通过逗号进行分隔，这样可以过滤更多的符合不同条件的元素。例如，在上面示例中，如果 filter()方法写成如下样式：

```
$(function(){
    $("div").filter(".red,.blue").css("background-color","red");
})
```

则将匹配到文档中<div class="blue">、<div class="red">和<div class="red pos"> 3 个标签，并设置它们的背景色为红色。

【示例 4】在下面示例中使用 filter()方法从\$("p")所匹配的 p 元素集合中过滤出包含两个 span 子元素的标签，然后为这些元素定义红色背景。演示效果如图 3.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").filter(function(index) {
        return $("span", this).length == 2;
    }).css("background-color","red");
})
</script>
<title>上机练习</title>
</head>
<body>
<p><span>第一段文本</span></p>
<p><span>第二段文本</span><span></span></p>
<p>第三段文本</p>
</body>
</html>
```

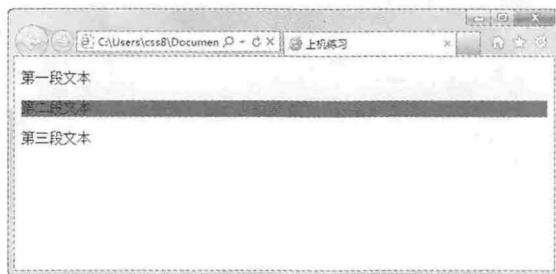


图 3.3 使用 filter()方法过滤元素

【代码详解】

filter()方法所包含的参数函数能够返回一个布尔值,在这个函数内部将对每个元素计算一次,工作原理类似\$.each()方法,如果调用的这个参数函数返回 false,则这个元素被删除,否则就会保留。在上面示例中,\$("span",this)将匹配当前元素内部的所有 span 元素,然后计算它的长度,检测如果当前元素包含了两个 span 元素,则返回 true,否则返回 false。filter()方法将根据参数返回值决定是否保留每个匹配元素。

在这个参数函数中包含一个 index 参数(默认的),该参数存储当前对象在 jQuery 对象中的下标位置,在函数体内,this 关键字指向当前元素对象,而不是 jQuery 对象。

由于参数函数可以实现各种复杂的计算和处理,所以使用 filter(fn)比 filter(expr)更为灵活,用户可以在参数函数中完成各种额外的任务,或者为每个元素添加附加行为和操作。

2. has()方法

has()方法也是一个过滤工具,不过它没有 filter()方法功能强大,但是使用敏捷,更方便用户把它作为专业过滤工具,专门负责保留包含特定后代的元素,去掉那些不含有指定后代的元素。

has()方法将会从给定的 jQuery 对象中重新创建一组匹配的对象。具体用法如下:

has(expr)

参数 expr 可以是一个 jQuery 选择器表达式字符串,也可以是一个元素或者一组元素。提供的选择器会一一测试原先那些对象的后代,含有匹配后代的对象将得以保留。也就是说,如果元素包含了与 expr 表达式相匹配的子元素,则将保留该元素,否则就会删除该元素。

【示例 5】使用 has()方法从\$("p")所匹配的 p 元素集合中过滤出包含类名为 red 的 span 子元素的标签,然后为这些元素定义红色背景。演示效果如图 3.4 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").has("span.red").css("background-color","red");
})
</script>
<title>上机练习</title>
</head>
<body>
<p><span class="red">第一段文本</span></p>
<p><span>第二段文本</span><span></span></p>
<p>第三段文本</p>
</body>
</html>
```

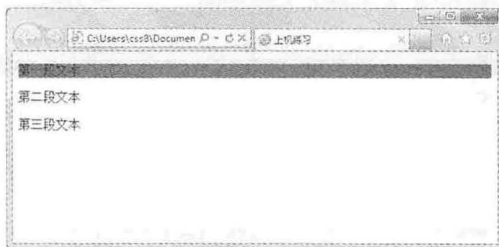


图 3.4 使用 has()方法过滤元素

3.1.4 判断

表达式判断的方法不直接过滤元素，仅作为一个检测工具为其他代码判断当前 jQuery 对象是否包含特定符合条件的元素提供方便。具体用法如下：

`is(expr)`

参数 `expr` 为一个 jQuery 选择器表达式，用来筛选符合特定条件的元素。

该方法的工作原理是：用一个表达式来检查当前选择的元素集合，如果其中至少有一个元素符合这个给定的表达式就返回 `true`。如果没有元素符合，或者表达式无效，就返回 `false`。实际上，`filter()`方法内部也是在调用这个函数，所以，`filter()`方法原有的规则在这里也适用。

【示例 6】 在下面示例中使用 `has()`方法检测 `$(“div”)`所匹配的 `div` 元素集合中是否包含 `span` 元素，如果包含则进行提示，否则就忽略。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    if($(".div").has("p"))
        alert("当前 jQuery 对象中包含有 span 子元素");
})
</script>
<title>上机练习</title>
</head>
<body>
<div class="blue"></div>
<div class="red"> </div>
<div class="green"></div>
<div class="red pos"> </div>
</body>
</html>
```

3.1.5 映射

通常情况下，映射是指两个元素集合之间元素相互“对应”的关系，实际上映射是一种间接引用。`map()`方法通过映射关系，把 jQuery 对象中每个元素映射到一个数组中，也就是说将一组元素转换成其他数组（不论是否是元素数组）。具体用法如下：

`map(callback)`

参数 `callback` 表示一个回调函数，将给每个元素执行的函数。用户可以用这个函数来建立一个列表，不论是值、属性还是 CSS 样式，或者其他特别形式，都可以用 `$.map()`方法建立。

【示例 7】 通过 `map()`方法把所有匹配的 `input` 元素的 `value` 属性值映射为一个新集合，然后调用 `get()`方法把集合内的数据转换为数组，再调用数组的 `join()`方法把集合元素连接为字符串，最后调用 jQuery 的 `append()`方法把这个字符串附加到 `<p>`标签中的末尾。演示效果如图 3.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```



```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").append( $("input").map(function(){
        return $(this).val();
    }).get().join("、 ") );
})
</script>
<title>上机练习</title>
</head>
<body>
<p><b>注册信息:</b></p>
<form>
    <input type="text" name="name" value="用户名"/>
    <input type="text" name="password" value="密码"/>
    <input type="text" name="url" value="http://www.baidu.com"/>
</form>
</body>
</html>

```

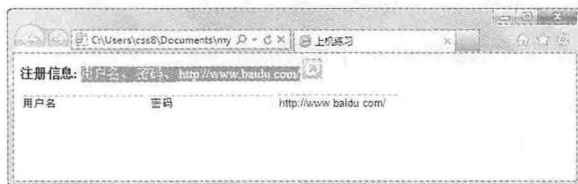


图 3.5 映射的应用效果

【提示】

map()方法根据调用的 jQuery 作为映射原对象,返回的结果也是一个 jQuery 对象,不过映射后的 jQuery 对象并非都是元素集合,也可能是其他任意类型的数据,这主要根据 map()的参数函数返回值来确定。例如,在上面示例中,如果修改为下面的代码形式:

```

$(function(){
    $("p").append( $("input").map(function(){
        return $(this).val();
    })[0] );
})

```

也就是说,让参数函数返回值为当前元素的值,则完成映射后的 jQuery 包含的元素也是一个值列表,在上面代码中 \$("input").map(function(){ return \$(this).val(); })[0] 返回值为一个字符串,并被附加到 <p> 标签中,如图 3.6 所示。

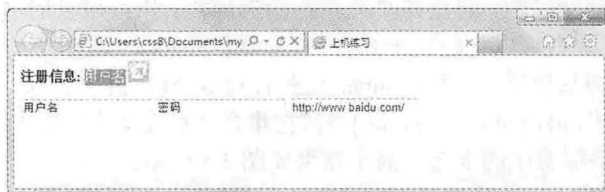


图 3.6 映射的值

3.1.6 清洗

与 eq()、filter()、has() 方法的过滤思路不同, not() 方法执行的是反向操作, 它能够从 jQuery 对象中删除符合条件的元素, 并返回这个清洗后的 jQuery。具体用法如下:

not(expr)

参数 expr 表示一个 jQuery 选择器表达式字符串, 当然也可以是一个元素或者多个元素。

【示例 8】通过 not() 方法排除首页导航菜单, 然后为其他菜单项定义统一的样式。演示效果如图 3.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#menu li").not(".home").css("color","red");    //清除 home 类菜单项
})
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
    <li class="home">首页</li>
    <li>论坛</li>
    <li>微博</li>
    <li>团购</li>
    <li>博客</li>
</ul>
</body>
</html>
```

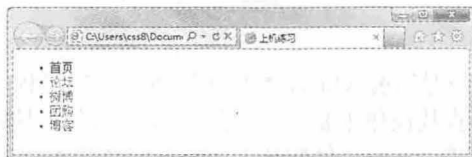


图 3.7 清洗效果

3.1.7 截取

jQuery 模仿 Array 对象的 slice() 方法也定义了一个 slice() 方法, 用来在当前 jQuery 对象中截取部分元素, 并把这个被截取的元素集合装在一个新的 jQuery 对象中返回。具体用法如下:

slice(start,[end])

参数 start 和 end 都是一个整数, 其中 start 表示开始选取子集的位置, 第 1 个元素是 0, 如果该参数为负数, 则表示从集合的尾部开始选起。end 是一个可选参数, 表示结束选取的位置, 如果不指定, 则表示到集合的结尾, 但是被截取的元素中不包含 end 所指定位置的元素。

【示例 9】通过 slice() 方法截取第 3、4 个菜单项, 然后为其他菜单项定义统一的样式。演示效果如图 3.8

所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("#menu li").slice(2,4).css("color","red"); //截取第 3、4 个菜单项
})
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
    <li class="home">首页</li>
    <li>论坛</li>
    <li>微博</li>
    <li>团购</li>
    <li>博客</li>
</ul>
</body>
</html>
```



图 3.8 片段截取

3.2 查 找

3.1 节所介绍的过滤操作主要是对当前 jQuery 对象进行再加工的过程，以便实现二次筛选，从而达到缩小匹配范围的目的。本节所介绍的查找操作主要是以当前 jQuery 对象为基础，查找父级、同级或者下级相关元素，以便实现延伸筛选，从而增强对文档的控制能力。jQuery 查找方法主要包括 16 种，详细说明如表 3.2 所示。

表 3.2 jQuery 查找方法

查 找 方 法	说 明
add(expr,[context])	把与表达式匹配的元素添加到 jQuery 对象中
children([expr])	取得一个包含匹配的元素集合中每一个元素的所有子元素的元素集合
closest(expr,[context])	从元素本身开始，逐级向上级元素匹配，并返回最先匹配的元素
contents()	查找匹配元素内部所有的子节点（包括文本节点）
find(expr)	搜索所有与指定表达式匹配的元素
next([expr])	取得一个包含匹配的元素集合中每一个元素紧邻的后面同辈元素的元素集合
nextAll([expr])	查找当前元素之后所有的同辈元素

续表

查 找 方 法	说 明
<code>nextUntil([selector])</code>	查找当前元素之后所有的同辈元素，直到遇到匹配的那个元素为止
<code>offsetParent()</code>	返回第一个匹配元素用于定位的父节点
<code>parent([expr])</code>	取得一个包含所有匹配元素的唯一父元素的元素集合
<code>parents([expr])</code>	取得一个包含所有匹配元素的祖先元素的元素集合（不包含根元素）
<code>parentsUntil([selector])</code>	查找当前元素的所有的父辈元素，直到遇到匹配的那个元素为止
<code>prev([expr])</code>	取得一个包含匹配的元素集合中每一个元素紧邻的前一个同辈元素的元素集合
<code>prevAll([expr])</code>	查找当前元素之前所有的同辈元素
<code>prevUntil([selector])</code>	查找当前元素之前所有的同辈元素，直到遇到匹配的那个元素为止
<code>siblings([expr])</code>	取得一个包含匹配的元素集合中每一个元素的所有唯一同辈元素的元素集合

3.2.1 向下查找后代元素

DOM 提供了以下 3 种访问后代元素的途径：

- ☑ 使用 `childNodes` 集合属性，这是一个所有元素都有的属性，这个集合包含了所有子节点，包括元素、文本节点和注释文本节点等。通过该集合可以遍历到所有的子元素。
- ☑ 使用 `firstChild` 和 `lastChild` 属性可以查找第 1 个和最后一个子元素。
- ☑ 使用 `getElementsByTagName()` 和 `getElementById()` 方法可以获取后代元素。

当然，jQuery 在这些基本途径基础上封装了形式各异的方法，使用这些方法能够方便、快速地找到所需要的不同的后代元素。

1. `children()` 方法

`children()` 方法能够查找当前元素的所有或者部分子元素，实际上它是 `childNodes` 集合的另一种用法。

具体用法如下：

```
children([expr])
```

参数 `expr` 表示 jQuery 选择器表达式字符串，用以过滤子元素。该参数为可选，如果省略，则将匹配所有的子元素，功能类似于 DOM 的 `childNodes`，不过返回值形式不同。`children()` 方法返回的是一个 jQuery，而 `childNodes` 返回的是一个数组。

【示例 10】 为当前列表结构的所有列表项定义一个下划线样式。演示效果如图 3.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#menu").children().css("text-decoration","underline");
})
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
<li class="home">首页</li>
```

```

<li>论坛</li>
<li>微博</li>
<li>团购</li>
<li>博客</li>
</ul>
</body>
</html>

```

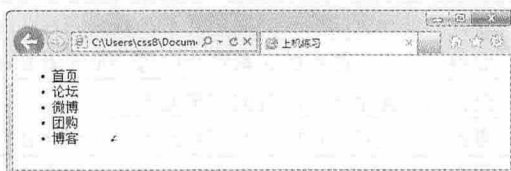


图 3.9 查找所有子元素

【示例 11】 在示例 10 中, 如果为 `children()` 方法添加一个表达式, 则可以在子元素中进行适当过滤。使用以下代码能够获取包含 `home` 类型的子元素。

```

$(function(){
    $("#menu").children(".home").css("text-decoration","underline");
})

```

上面代码也可以这样写:

```

$(function(){
    $("#menu li").not(".home").css("text-decoration","underline");
})

```

2. contents()方法

与 `children()` 方法相似, `contents()` 方法也是用来查找子内容的, 但它不仅获取子元素, 还可以获取文本节点、注释节点等, 因此读者可以把它视为 DOM 中 `childNodes` 属性的 jQuery 实现。具体用法如下:

`contents()`

该方法没有参数, 功能等同于 DOM 的 `childNodes`, 不过返回值形式不同。`contents()` 方法返回的是一个 jQuery 对象, 而 `childNodes` 返回的是一个数组。

3. find()方法

`find()` 方法与 `children()` 方法相似, 都是用来向下查找元素的, 但是 `find()` 方法能够查找所有后代元素, 而 `children()` 方法仅能够查找子元素。与 `contents()` 方法相比, 两者查找的范围存在交集, `find()` 查找的种类仅限于元素, 而 `contents()` 方法不仅查找元素, 还可以查找文本节点、注释节点等。

【示例 12】 使用 `jQuery()` 函数获取页面中 `body` 的子元素 `div`, 然后分别调用 `children()` 和 `find()` 方法获取其包含的所有 `div` 元素, 同时使用 `contents()` 获取其包含的节点。在浏览器中预览, 则可以看到 `children("div")` 包含 3 个元素, `find("div")` 返回 5 个元素, 而 `contents()` 返回 7 个元素, 其中包含两个文本节点。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var j = $("body > div");
    alert(j.children("div").length);    //返回 3 个 div 元素

```



```

    alert(j.find("div").length);      //返回 5 个 div 元素
    alert(j.contents().length);      //返回 7 个元素, 包括 5 个 div 元素, 2 个文本节点 (空格)
  })
</script>
<title>上机练习</title>
</head>
<body>
<div>
  <div>
    <div></div>
    <div> </div>
  </div>
  <div></div>
  <div></div>
</div>
</body>
</html>

```

3.2.2 向上查找祖先元素

DOM 提供了一种访问祖先元素的途径——使用 `parentNode` 属性访问父元素。在这方面, jQuery 提供了更加丰富和强大的方法, 方便用户访问不同类型和级别的祖先元素。

1. `parents()`方法

`parents()`方法能够查找所有匹配元素的祖先元素。具体用法如下:

`parents([expr])`

参数 `expr` 表示 jQuery 选择器表达式字符串, 用以过滤祖先元素。该参数为可选, 如果省略, 则将匹配所有元素的祖先元素。

【示例 13】 查找所有匹配的 `img` 元素的祖先元素, 并为它们定义统一的边框样式。演示效果如图 3.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
  $("img").parents().css({"border":"solid 1px red","margin":"10px"});
  alert($("img").parents().length);      //返回 4, 分别是 span、div、body 和 html
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
  <span>
    
  </span>
  

```



```

</div>
</body>
</html>

```

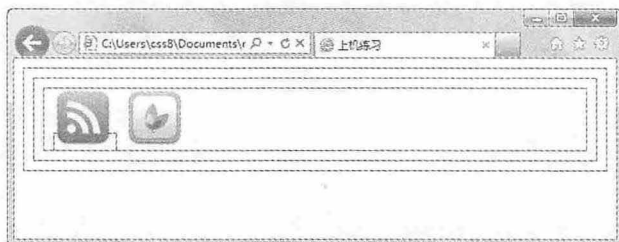


图 3.10 查找所有祖先元素

【提示】

parents()方法将查找所有匹配元素的祖先元素，如果存在重合的祖先元素，则仅记录一次。当然也可以在 parents()参数中定义一个过滤表达式，过滤出符合条件的祖先元素。

2. parent()方法

parent()方法是对 parents()方法的延伸，它可以取得一个包含所有匹配元素的唯一父元素的元素集合。具体用法如下：

parents([expr])

参数 expr 表示 jQuery 选择器表达式字符串，用以过滤父元素。该参数可选，如果省略，则将匹配所有元素的唯一父元素。

【示例 14】 针对示例 13，将 parents()方法替换为 parent()方法，将查找所有匹配的 img 元素的父元素，并为它们定义统一的边框样式。演示效果如图 3.11 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("img").parent().css({"border":"solid 1px red","margin":"10px"});
    $("img").parent().each(function(){alert(this.nodeName)});    //提示 SPAN 和 DIV 元素
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <span>
        
    </span>
    
</div>
</body>
</html>

```



图 3.11 查找所有父元素

3. parentsUntil()方法

parentsUntil()方法是对 parents()方法的一个有益补充，它可以查找指定范围的所有祖先元素，相当于在 parents()方法返回集合中截取部分祖先元素。具体用法如下：

parentsUntil([selector])

参数 selector 表示 jQuery 选择器表达式字符串，用以确定范围的祖先元素。该参数为可选，如果省略，则将匹配所有祖先元素。

【示例 15】 在以下示例中 \$('li.l31') 将匹配三级菜单下的第 1 个列表项，然后使用 parentsUntil('.u1') 方法获取它的所有祖先元素，但是只包含 <ul class="u1"> 标签范围内的元素，最后为查找的祖先元素定义边框样式。演示效果如图 3.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $('li.l31').parentsUntil('.u1').css({"border":"solid 1px red","margin":"10px"});
})
</script>
<title>上机练习</title>
</head>
<body>
<ul class="u1">一级菜单
    <li class="l1">1</li>
    <li class="l2">2
        <ul class="u2">二级菜单
            <li class="l21">21</li>
            <li class="l22">22
                <ul class="u3">三级菜单
                    <li class="l31">31</li>
                    <li class="l32">32</li>
                    <li class="l33">33</li>
                </ul>
            </li>
            <li class="item-c">C</li>
        </ul>
    </li>
    <li class="l3">3</li>
</ul>
</body>
</html>
```

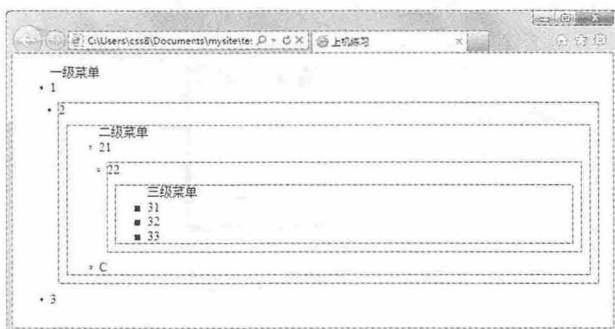



图 3.12 查找指定范围的祖先元素

【提示】

如果 `parentsUntil()` 方法没有包含参数，则将查找所有的祖先元素，此时 `parentsUntil()` 方法与 `parents()` 方法查找的结果是相同的。例如，针对上面示例，不给 `parentsUntil()` 方法传递参数，则查找的效果如图 3.13 所示。

```
$(function(){
    $('li.l31').parentsUntil().css({"border":"solid 1px red","margin":"10px"});
})
```



图 3.13 查找所有祖先元素

4. `offsetParent()` 方法

`offsetParent()` 方法能够查找到当前元素的第一个定位祖先元素，使用该方法可以快速找到当前元素的定位框，并据此设置元素的绝对位置。具体用法如下：

`offsetParent()`

该方法没有参数，使用该方法可以返回包含祖先元素中第 1 个定位元素的 jQuery 对象（定位元素就是元素的 `position` 属性被设置为 `relative` 或者 `absolute` 的元素）。`offsetParent()` 方法仅对可见元素有效。

5. `closest()` 方法

与 `offsetParent()` 方法一样，`closest()` 也是一个特殊的方法，用来查找指定的父元素。它主要为事件处理而设计，处理事件委派非常有用。具体用法如下：

`closest(expr,[context])`

参数 `expr` 可以是字符串，也可以是数组，用以过滤元素的表达式。也可以传递一个字符串数组，用于查找多个元素。`context` 是一个可选参数，表示一个元素，用来设置待查找的 DOM 元素集、文档或 jQuery 对象。如果省略，则表示待查找所有祖先元素。

`closest()` 方法被解析时，会首先检查当前元素是否匹配，如果匹配则直接返回元素本身。如果不匹配则向上查找父元素，一层一层往上，直到找到匹配选择器的元素。如果什么都没找到则返回一个空的 jQuery 对象。

closest()方法与 parents()方法不同,主要区别如下:

- ☑ closest()方法从当前元素开始匹配寻找,而 parents()方法从父元素开始匹配寻找。
- ☑ closest()方法逐级向上查找,直到发现匹配的元素后就停止了,而 parents()方法一直向上查找直到根元素,然后把这些元素放进一个临时集合中,再用给定的选择器表达式过滤。
- ☑ closest()方法返回 0 或 1 个元素,而 parents()方法可能返回 0 个、1 个,或者多个元素。

【示例 16】在以下示例中\$(“li.l31”)将匹配三级菜单下的第一个列表项,然后使用 closest(“ul”)方法获取祖先元素中最靠近当前元素的父元素,最后为这个元素定义边框样式。演示效果如图 3.14 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(“li.l31”).closest(“ul”).css({“border”:“solid 1px red”,“margin”:“10px”});
})
</script>
<title>上机练习</title>
</head>
<body>
<ul class=“u1”>一级菜单
    <li class=“l1”>1</li>
    <li class=“l2”>2
        <ul class=“u2”>二级菜单
            <li class=“l21”>21</li>
            <li class=“l22”>22
                <ul class=“u3”>三级菜单
                    <li class=“l31”>31</li>
                    <li class=“l32”>32</li>
                    <li class=“l33”>33</li>
                </ul>
            </li>
            <li class=“item-c”>C</li>
        </ul>
    </li>
    <li class=“l3”>3</li>
</ul>
</body>
</html>
```

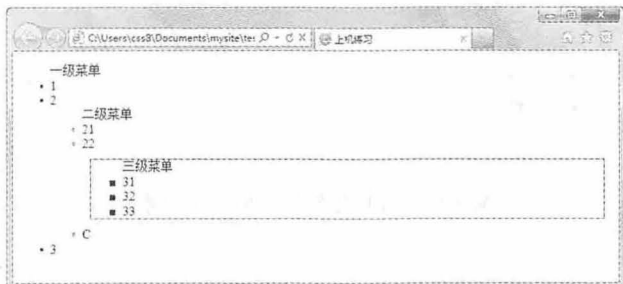


图 3.14 查找指定的父元素

3.2.3 向上查找兄弟元素

在 DOM 中可以使用 `previousSibling` 属性查找匹配元素的前一个兄弟节点。在这方面, jQuery 提供了更加丰富和强大的方法, 方便用户访问不同类型的兄弟元素。

1. `prev()`方法

`prev()`与 `previousSibling` 属性功能相同, 但是 `previousSibling` 属性还可以匹配上一个相邻的任意类型的节点, 如文本节点等, 而 `prev()`方法仅能够匹配上一个相邻的元素。具体用法如下:

`prev([expr])`

参数 `expr` 表示 jQuery 选择器表达式字符串, 用以过滤匹配元素。该参数为可选, 如果省略, 则将匹配所有上一个相邻的元素。

【示例 17】 查找类名为 `red` 的 `p` 元素, 然后使用 `prev()`方法查找它的上一个相邻的 `p` 元素, 并为其定义边框样式。演示效果如图 3.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".red").prev().css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章 </h2>
<p class="blue">少小离家老大回, </p>
<p>乡音无改鬓毛衰。</p>
<p class="red">儿童相见不相识, </p>
<p>笑问客从何处来。</p>
</body>
</html>
```

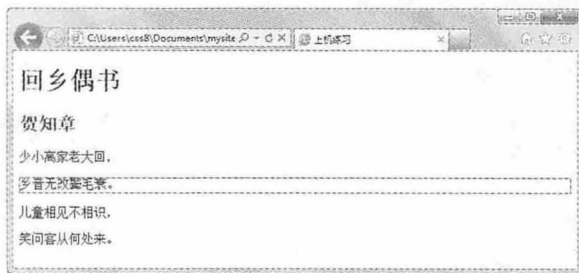


图 3.15 查找相邻的上面一个元素

2. `prevAll()`方法

`prevAll()`方法比 `prev()`方法查找的范围要大, 不仅包括相邻的兄弟元素, 还包括所有同辈元素。具体用

法如下:

`prevAll [expr]`

参数 `expr` 表示 jQuery 选择器表达式字符串, 用以过滤匹配元素。该参数为可选, 如果省略, 则将匹配所有上面同辈元素。

【示例 18】查找类名为 `red` 的 `p` 元素, 然后使用 `prevAll()` 方法查找它的上面同辈的所有 `p` 元素, 并为它定义边框样式。演示效果如图 3.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".red").prevAll("p").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章</h2>
<p class="blue">离别家乡岁月多, </p>
<p>近来人事半消磨。</p>
<p class="red">惟有门前镜湖水, </p>
<p>春风不改旧时波。</p>
</body>
</html>
```

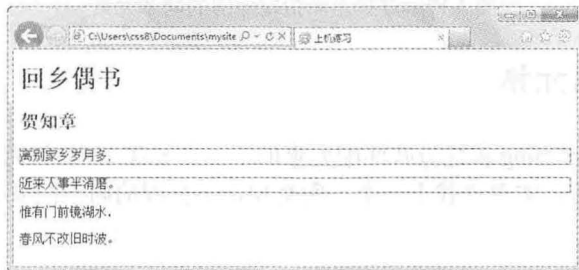


图 3.16 查找上面所有同辈元素

3. `prevUntil()` 方法

`prevUntil()` 方法介于 `prevAll()` 方法和 `prev()` 方法之间。`prevAll()` 方法选取前面所有的同辈元素, `prev()` 方法选取前面紧邻的同辈元素, 而 `prevUntil()` 方法能够选取指定范围的相邻同辈元素。具体用法如下:

`prevUntil([selector])`

参数 `selector` 表示 jQuery 选择器表达式字符串, 用以过滤匹配元素。该参数为可选, 如果省略, 则将匹配所有上面同辈元素。

【示例 19】查找类名为 `red` 的 `p` 元素, 然后使用 `prevUntil("h1")` 方法查找 `h1` 元素前面的所有同辈元素, 并为它定义边框样式。演示效果如图 3.17 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```



```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".red").prevUntil("h1").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章</h2>
<p class="blue">离别家乡岁月多,</p>
<p>近来人事半消磨.</p>
<p class="red">惟有门前镜湖水,</p>
<p>春风不改旧时波.</p>
</body>
</html>

```

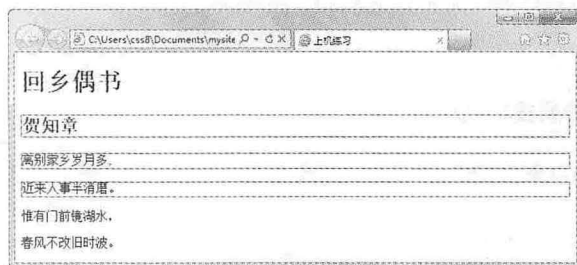


图 3.17 查找上面指定范围的同辈元素

3.2.4 向下查找兄弟元素

在 DOM 中可以使用 `nextSibling` 属性查找匹配元素的下一个兄弟节点。与向上查找兄弟元素一样, jQuery 也提供了 3 个类似的操作方法, 实现查找下一个、所有和指定范围的向下同辈元素。

1. `next()`方法

`next()`方法与 `nextSibling` 属性功能相同, 但是 `nextSibling` 属性还可以匹配下一个相邻的任意类型的节点, 如文本节点等, 而 `next()`方法仅能够匹配下一个相邻的元素。具体用法如下:

`next([expr])`

参数 `expr` 表示 jQuery 选择器表达式字符串, 用以过滤匹配元素。该参数为可选, 如果省略, 则将匹配所有下一个相邻的元素。

【示例 20】 查找类名为 `red` 的 `p` 元素, 然后使用 `next()`方法查找它的下一个相邻的 `p` 元素, 并为它定义边框样式。演示效果如图 3.18 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>

```

```

<script type="text/javascript" >
$(function(){
    $(".red").next("p").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章 </h2>
<p class="blue">少小离家老大回, </p>
<p>乡音无改鬓毛衰。</p>
<p class="red">儿童相见不相识, </p>
<p>笑问客从何处来。</p>
</body>
</html>

```

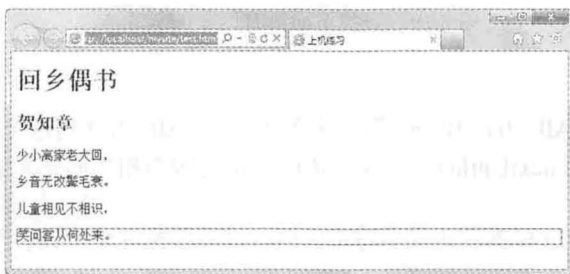


图 3.18 查找相邻的下面一个元素

2. nextAll()方法

nextAll()方法比 next()方法查找的范围要大, 不仅包括相邻的兄弟元素, 还包括所有同辈元素。具体用法如下:

nextAll [expr]

参数 expr 表示 jQuery 选择器表达式字符串, 用以过滤匹配元素。该参数为可选, 如果省略, 则将匹配所有下面的同辈元素。

【示例 21】查找类名为 blue 的 p 元素, 然后使用 nextAll()方法查找它的下面同辈的所有 p 元素, 并为它们定义边框样式。演示效果如图 3.19 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $(".blue").nextAll("p").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>

```

```

<h2>贺知章 </h2>
<p class="blue">少小离家老大回, </p>
<p>乡音无改鬓毛衰。</p>
<p class="red">儿童相见不相识, </p>
<p>笑问客从何处来。</p>
</body>
</html>

```

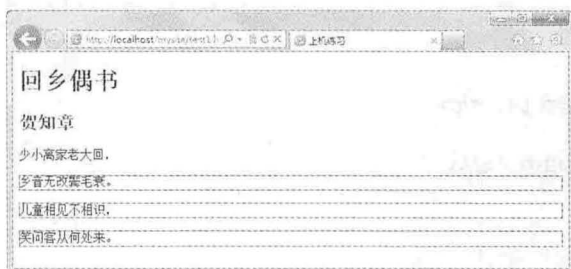


图 3.19 查找下面所有同辈的 p 元素

3. nextUntil()方法

nextUntil()方法介于 nextAll()方法和 next()方法之间。nextAll()方法选取下面所有同辈元素，next()方法选取下面紧邻的同辈元素，而 nextUntil()方法能够选取指定范围的相邻同辈元素。具体用法如下：

nextUntil([selector])

参数 selector 表示 jQuery 选择器表达式字符串，用以过滤匹配元素。该参数为可选，如果省略，则将匹配所有下面同辈元素。

【示例 22】查找类名为 blue 的 p 元素，然后使用 nextUntil(".red")方法查找类名为 red 的元素前面的所有同辈元素，并为其定义边框样式。演示效果如图 3.20 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".blue").nextUntil(".red").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章 </h2>
<p class="blue">少小离家老大回, </p>
<p>乡音无改鬓毛衰。</p>
<p class="red">儿童相见不相识, </p>
<p>笑问客从何处来。</p>
</body>
</html>

```




图 3.20 查找下面指定范围的同辈元素

3.2.5 查找兄弟元素

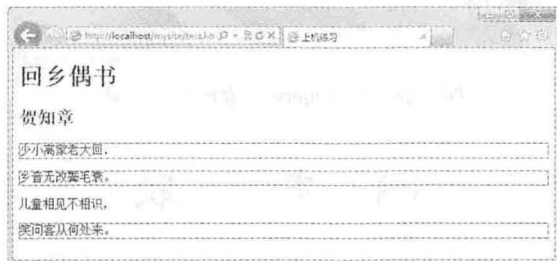
除了向上和向下查找兄弟元素，还可以使用 `siblings()` 方法查找所有兄弟元素，不管其位置在前还是在后。具体用法如下：

`siblings([expr])`

参数 `expr` 表示 jQuery 选择器表达式字符串，用以过滤匹配元素。该参数为可选，如果省略，则将匹配所有同辈兄弟元素。

【示例 23】查找类名为 `red` 的 `p` 元素，然后使用 `siblings("p")` 方法查找所有同辈的 `p` 元素，并为它定义边框样式。演示效果如图 3.21 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".red").siblings("p").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章 </h2>
<p class="blue">少小离家老大回，</p>
<p>乡音无改鬓毛衰。</p>
<p class="red">儿童相见不相识，</p>
<p>笑问客从何处来。</p>
</body>
</html>
```

图 3.21 查找所有同辈的 `p` 元素

3.2.6 添加查找对象

如果用户对查找的结果并不满意,还可以使用 `add()` 方法向查找的结果集中添加新的查找内容。具体用法如下:

`add(expr,[context])`

☑ 参数 `expr` 表示 jQuery 选择器表达式字符串,用于匹配元素并添加表达式字符串,或者用于动态生成的 HTML 代码,如果是一个字符串数组则返回多个元素。

☑ `context` 为可选参数,可以是待查找的 DOM 元素集、文档或 jQuery 对象。

【示例 24】查找类名为 `red` 的 `p` 元素,然后使用 `siblings("p")` 方法查找所有同辈的 `p` 元素,再使用 `add("h1,h2")` 方法,把一级标题和二级标题也添加到当前 jQuery 对象中,最后为新的 jQuery 内所有元素定义边框样式。演示效果如图 3.22 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".red").siblings("p").add("h1,h2").css("border","solid 1px red");
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>回乡偶书</h1>
<h2>贺知章</h2>
<p class="blue">少小离家老大回,</p>
<p>乡音无改鬓毛衰.</p>
<p class="red">儿童相见不相识,</p>
<p>笑问客从何处来.</p>
</body>
</html>
```

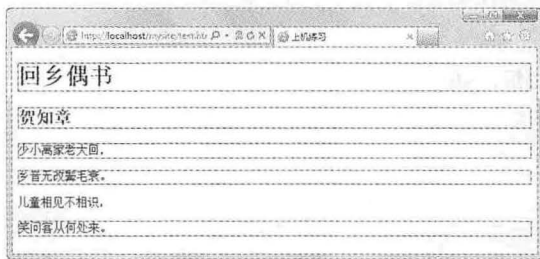


图 3.22 为 jQuery 对象添加新元素

3.3 串 联

jQuery 语法的最大亮点就是链式语法,它能够实现在一行代码中完成各种复杂的任务。这种连续书写

的代码显得灵巧、优雅，因此称之为链式语法。不过这种语法有时候会感觉不方便，因为很多方法通过点运算符串联在一起，操作的 jQuery 对象不能够前后照应，但是很多情况下我们希望用不同的方法操作不同的 jQuery 对象，或者前后方法能够相互影响，为此 jQuery 提供了两个串联专用工具——`andSelf()`和 `end()`。

3.3.1 绑定前后 jQuery 对象

下面先看一个简单的示例：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").find("p").css({"border":"solid 1px red","margin":"4px"});
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <p>第 1 段文本</p>
    <p>第 2 段文本</p>
</div>
</body>
</html>
```

在上面代码中，`$("div")`匹配文档中的 `<div>` 标签，然后调用 jQuery 对象的 `find()` 方法获取 `<div>` 标签包含的两个 `<p>` 标签，此时 `$("div")` 和 `$("div").find("p")` 就属于两个不同的 jQuery 对象。如果希望后面的 CSS 样式同时影响到 `<div>` 和 `<p>` 标签，就比较困难，此时在浏览器中预览，可以看到仅 `<p>` 标签显示边框样式，如图 3.23 所示。

如果希望同时为外围的 `<div>` 标签也定义相同的样式，则最简单的方法是重新书写一行代码，单独为 `div` 元素定义样式。显然这种做法与 jQuery 的链式语法设计原则是相违背的。

为此，jQuery 定义了 `andSelf()` 方法帮助用户把前后 jQuery 对象串联在一起，形成一个新的 jQuery 对象。例如，针对上面示例，在 `find("p")` 后面添加 `andSelf()` 方法，把 `$("div")` 和 `find("p")` 两个不同的 jQuery 对象链接在一起，最后再为它们定义统一的样式。代码如下：

```
$(function(){
    $("div").find("p").andSelf().css({"border":"solid 1px red","margin":"4px"});
})
```

此时在浏览器中预览，效果如图 3.24 所示。

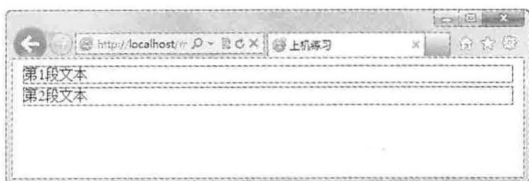


图 3.23 匹配 p 元素的 jQuery

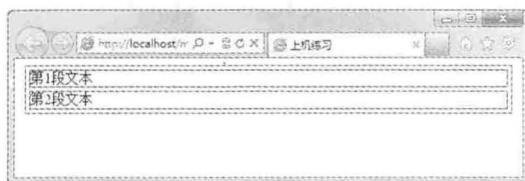


图 3.24 andSelf() 方法的应用

提示，对于筛选或查找后的元素，加入先前所选元素时使用的 `andSelf()` 方法将会很有用。

3.3.2 返回前一个 jQuery 对象

继续以下面这个示例为例进行说明：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").find("p").css({"border":"solid 1px red","margin":"4px"});
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <p>第 1 段文本</p>
    <p>第 2 段文本</p>
</div>
</body>
</html>
```

如果希望为<p>标签定义边框样式，然后再为<div>标签定义背景色，那么简单的做法就是重新换一行为<div>标签定义样式。

不过现在利用 jQuery 定义的 end() 方法，可以保持在一行内完成两行任务，即调用 find("p").css() 后，再调用 end() 方法返回 \$("div") 方法匹配的 jQuery 对象，而不是 find() 方法所查找的 jQuery。代码如下：

```
$(function(){
    $("div").find("p").css({"border":"solid 1px red","margin":"4px"}).end().css({"background":"blue","color":"white",
    "padding":"4px"});
})
```

在上面代码中，首先为 \$("div").find("p") 定义的 jQuery 所包含的元素定义边框样式，然后调用 end() 方法，返回上一次匹配的 jQuery 对象，即 \$("div") 定义的 jQuery 对象，再为该对象调用 css() 方法定义背景样式，最后显示效果如图 3.25 所示。

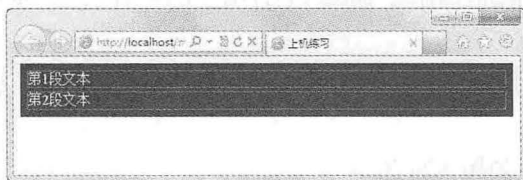


图 3.25 end() 方法的应用

第 4 章

DOM 操作

( 视频讲解：2 小时 10 分钟)

DOM 是 Document Object Model 的缩写，中文翻译为文档对象模型，根据 W3C DOM 规范 (<http://www.w3.org/DOM/>)，DOM 是一种与浏览器、平台、语言无关的接口，使用该接口可以访问页面内的所有节点对象。实际上，DOM 是一套对文档内容进行抽象和概念化的方法。

在早期 JavaScript 版本中就已经提供了初级的 DOM，它向程序员提供了对 Web 文档的某些实际内容进行查询和操作的方法，后来这种初级方法统称为 0 级 DOM (DOM Level 0)。1998 年，W3C 组织推出了一份标准 DOM，即所谓的 1 级 DOM (<http://www.w3.org/TR/REC-DOM-Level-1/>)，然后在 2000 年又推出了 2 级 DOM (<http://www.w3.org/TR/DOM-Level-2-Core/>)。

W3C DOM 标准包含很多模块，每个模块下面又包含很多子模块。其中用来操纵标记文档（如 HTML 和 XML 等）的模块被称为核心模块，即所谓的 DOM Core。

DOM Core 统一了访问网页文档的方法，并为不同类型的节点对象提供了统一的操作方法和属性。JavaScript 中的 `getElementById()`、`getElementsByTagName()`、`getAttribute()` 和 `setAttribute()` 等方法都是 DOM Core 模块的组成部分。

jQuery 作为一种 JavaScript 库，继承并优化了 JavaScript 访问 DOM 对象的特性，使开发人员能更加方便地操作 DOM 对象。

【提示】

为了能够更详细讲解 DOM 操作，我们需要构建一份网页文档，并提炼出它的文档结构，以 DOM 树形模式演示出来。HTML 结构如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>上机练习</title>
</head>
<body>
<h1>DOM</h1>
<p>DOM 实际上是以面向对象方式描述的文档模型。DOM 定义了表示和修改文档所需的对象、这些对象的行为和属性，以及这些对象之间的关系。可以把 DOM 理解为是页面上数据和结构的一个树形表示，不过页面当然并不是以这种树的方式具体实现。根据 W3C DOM 规范，DOM 是 HTML 与 XML 的应用编程接口 (API)，DOM 将整个页面映射为一个由层次节点组成的文件。包括 1 级、2 级、3 级共 3 个级别。</p>
<ul>
```

1 级 DOM 在 1998 年 10 月份成为 W3C 的提议, 由 DOM 核心与 DOM HTML 两个模块组成。DOM 核心能映射以 XML 为基础的文档结构, 允许获取和操作文档的任意部分。DOM HTML 通过添加 HTML 专用的对象与函数对 DOM 核心进行了扩展。

2 级 DOM 通过对象接口增加了对鼠标和用户界面事件 (DHTML 长期支持鼠标与用户界面事件)、范围、遍历 (重复执行 DOM 文档) 和层叠样式表 (CSS) 的支持。同时也对 DOM 1 的核心进行了扩展, 从而可支持 XML 命名空间。2 级 DOM 引进了几个新 DOM 模块来处理新的接口类型: DOM 视图, 描述跟踪一个文档的各种视图 (使用 CSS 样式设计文档前后) 的接口; DOM 事件, 描述事件接口; DOM 样式, 描述处理基于 CSS 样式的接口; DOM 遍历与范围, 描述遍历和操作文档树的接口。

3 级 DOM 通过引入统一方式载入和保存文档和文档验证方法对 DOM 进行进一步扩展, DOM 3 包含一个名为 "DOM 载入与保存" 的新模块, DOM 核心扩展后可支持 XML 1.0 的所有内容, 包括 XML Infoset、XPath 和 XML Base。

</body>

</html>

每一份网页文档都可以使用 DOM 来进行描述, 每一份 DOM 都会把网页看作是一棵节点树。浏览器在渲染文档时, 会自动在内存中构建一份完整的 DOM 树形结构 (或者说是树形结构列表), 如图 4.1 所示。

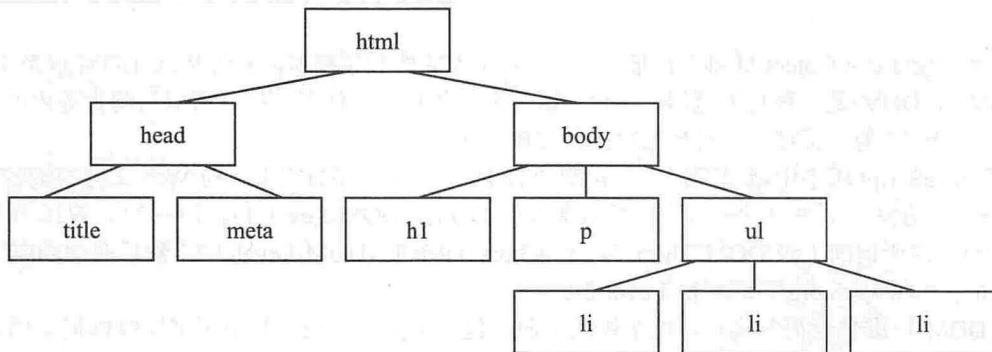


图 4.1 DOM 树形结构

而浏览器在窗口中所显示的网页内容如图 4.2 所示。

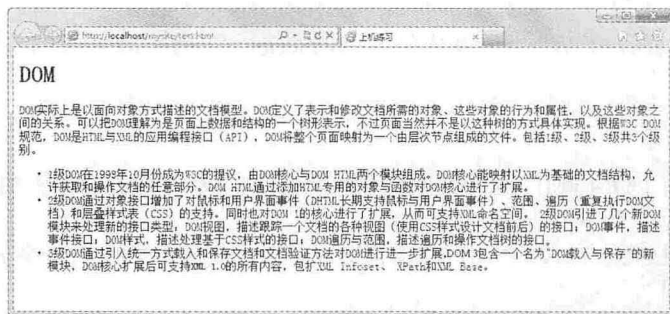


图 4.2 网页文档显示出来的效果

注意, W3C 并没有定义标准为 0 级的 DOM。所谓的 0 级 DOM, 它仅是 DOM 历史上的一个参考点, 实际上 0 级 DOM 被认为是 IE 4.0、Netscape Navigator 4.0 支持的最早的 DHTML 标准。

4.1 创建节点

根据 DOM 规范, 节点是 DOM 中有效而完整的结构中的最小单元, 包括元素、属性、文本、文档、注

释等。网页文档中读者可以把所有内容都归为某一类节点。在实际开发中,要创建动态内容,主要操作的节点包括元素、属性和文本。

4.1.1 创建元素

DOM 为 Document 对象定义了一个 `createElement()` 方法,该方法能够根据参数传递的标签字符串创建指定的元素对象,并返回新创建的元素对象。具体用法如下:

```
document.createElement(name)
```

返回对象仅在 JavaScript 上下文中有效,如果要把它添加到文档中,还需要调用 Element 对象的 `appendChild()` 方法实现。例如,先创建 div 元素对象,然后添加到文档中,代码如下:

```
<script type="text/javascript" >
window.onload = function(){           //页面初始化函数
    var div = document.createElement("div"); //创建 div 元素
    document.body.appendChild(div);       //把创建的 div 元素添加到 DOM 文档树中
}
</script>
```

jQuery 简化了这个操作,直接使用 jQuery 构造函数 `$()` 创建元素对象。具体用法如下:

```
$(html)
```

该函数能够根据参数 `html` 所传递的 HTML 标记字符串,创建一个 DOM 对象,并将该对象包装为 jQuery 对象返回。

【注意】

参数字符串必须符合严谨型 XHTML 结构的要求,标记应该包含起始标签和结束标签。如果没有结束标签,则应该添加闭合标记,即在起始标签中添加斜线。例如,下面字符串参数都是合法的:

```
"<h1></h1>"           //合法的参数字符串
"<h1 />"               //合法的参数字符串
```

而下面的字符串参数都是非法的:

```
"<h1>"                //非法的参数字符串
"</h1>"                //非法的参数字符串
```

然后,将创建的元素插入到文档中。动态创建的元素不会自动添加到文档中,需要使用其他方法把它添加到文档中。例如,可以使用 jQuery 的 `append()` 方法把创建的 div 元素添加到文档 body 元素节点下,代码如下:

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){           //页面初始化函数
    var $div = $("<div></div>"); //创建 div 对象
    $("body").append($div);     //把创建的 div 对象添加到文档中
})
</script>
```

在浏览器中运行代码后,新创建的 h1 元素被添加到文档中,由于该元素没有包含任何文本,所以看不到任何显示效果。

【提示】

jQuery 和 JavaScript 都用了两行代码在文档中创建元素,但 jQuery 稍显简便,仅用到复合函数 `$()` 和 jQuery 的 `append()` 方法。而 JavaScript 需要调用 Document 对象的 `createElement()` 方法和 Element 对象的 `appendChild()` 方法。从执行效率角度分析,两者差距明显,JavaScript 要比 jQuery 快 10 倍以上,在 IE 8 中差距会拉大到 30 倍以上,其他主流浏览器的执行效率差距就更大了。

4.1.2 输入文本

文本节点无法独立存在，必须包裹在元素节点内部，因此在创建文本之前，应确保新建或者选择元素节点。DOM 在 Document 对象中定义了 `createTextNode()` 方法，调用该方法可以创建文本节点。具体用法如下：

```
var txt = document.createTextNode("text");
```

该方法包含一个参数，即新建文本节点所包含的文本字符串，参数中不能够包含任何 HTML 标签，否则 JavaScript 会把这些标签作为字符串进行显示，最后返回新创建的文本节点。

新创建的文本节点不会自动增加到 DOM 结构树中，如果要把创建的文本节点添加到新建或指定元素内，需要使用 `appendChild()` 方法实现。例如，为 `div` 元素创建一行文本，并在文档中显示，代码如下：

```
window.onload = function(){
    var div = document.createElement("div");
    var txt = document.createTextNode("DOM");
    div.appendChild(txt);
    document.body.appendChild(div);
}
```

jQuery 创建文本节点比较简单，直接把文本字符串添加到元素标记字符串之中，然后使用 `append()` 等方法把它们添加到 DOM 文档结构树中。

【示例 1】 在文档中插入一个 `div` 元素，并在 `<div>` 标签中包含 DOM 的文本信息。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $div = $("<div>DOM</div>");
    $("body").append($div);
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

【分析】

从代码输入的角度分析，JavaScript 实现相对麻烦，用户需要分别创建元素节点和文本节点，然后把文本节点添加到元素节点中，再把元素添加到 DOM 结构树中。而 jQuery 经过包装之后，与 jQuery 创建元素节点操作相同，仅需要两步操作即可快速实现。

从执行效率角度分析，JavaScript 直接实现要比 jQuery 实现快 80 倍以上，在执行速度最慢的 IE 浏览器中，两者差距也在 10 倍以上。

4.1.3 设置属性

在 DOM 规范中，属性节点比较特殊，用户无法通过 Node 对象提供的方法遍历或者定位属性节点，必须使用 Element 对象定义的特定方法来创建和访问属性节点。DOM 为每个元素都定义了 `setAttribute()` 方法，

调用该方法可以创建属性节点，并设置属性节点包含的值。具体用法如下：

```
e.setAttribute(name,value);
```

其中，参数 `e` 表示指定的元素对象，`name` 和 `value` 参数分别表示属性名称和属性值。属性名称和属性值必须以字符串的形式进行传递。如果元素中存在指定的属性，它的值将被刷新；如果不存在，则 `setAttribute()` 方法将为元素创建该属性并赋值。例如，以示例 1 为例，调用 `setAttribute()` 方法为 `div` 元素设置一个 `title` 属性，实现代码如下：

```
window.onload = function(){
    var div = document.createElement("div");
    var txt = document.createTextNode("DOM");
    div.appendChild(txt);
    document.body.appendChild(div);
    div.setAttribute("title","盒子");           //为 div 元素定义 title 属性
}
```

jQuery 创建属性节点与创建文本节点类似，简单而又方便。例如，针对上面示例，可以在 jQuery 构造函数中以字符串形式简单设置。使用 jQuery 实现的完整代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function){
    var $div = $("<div title='盒子'>DOM</div>");
    $("body").append($div);
}
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

从上面示例代码可以看到，不管是创建元素节点、文本节点，还是属性节点，都可以模仿 HTML 具体用法编写 HTML 代码字符串，然后把这个字符串作为参数传递给 `$()` 函数即可，这大大降低了 DOM 操作的难度。

【分析】

从编写代码的角度分析，直接使用 JavaScript 实现需要单独为元素设置属性，而 jQuery 能够直接把元素、文本和属性混合在一起，根据 HTML 具体用法编写成字符串，再传递给 `$()` 函数即可。从执行效率角度分析，JavaScript 实现与 jQuery 实现的效率差距非常大。在 JavaScript 执行速度最快的 Safari 浏览器中循环执行 1000 次，JavaScript 耗时为十几毫秒，而 jQuery 耗时为 500 多毫秒。不同环境、版本和每次执行时间可能略有误差，但是差距基本保持在 40 倍左右。在执行速度最慢的 IE 浏览器中进行同比测试，则 JavaScript 耗时为 300~400 毫秒，而 jQuery 耗时为 3500 多毫秒，可见两者差距也在 10 倍左右。

由此可见，jQuery 以一种简易的方法代替繁琐的操作简化了 Web 开发的难度和门槛。当然，由于 jQuery 仅是在 JavaScript 基础上外部代码封装，所以执行速度并没有得到优化，相反却拖延了代码的执行效率。因此，在可能的情况下，建议读者结合使用 JavaScript 和 jQuery，以提高代码执行效率。

4.2 插入内容

jQuery 提供了众多在文档中插入内容的方法，这些方法从不同角度和用途进行设计，极大地方便了用户操作，使开发思维和开发速度提升到一个新的境界。

4.2.1 内部插入

JavaScript 提供了两个方法实现在节点内部插入元素，其中 `appendChild()` 方法与 jQuery 的 `append()` 方法相对应，`insertBefore()` 方法与 jQuery 的 `prepend()` 方法相对应。

`appendChild()` 方法能够把参数指定的元素插入到指定节点内的尾部。具体用法如下：

```
nodeObject.appendChild(newchild)
```

其中，`nodeObject` 表示节点对象。参数 `newchild` 表示要添加的子节点。插入成功之后，返回这个插入的节点。

【示例 2】 演示如何把一个 `h1` 元素添加到 `div` 元素内部的尾部，动态插入的结构如图 4.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var h1 = document.createElement("h1");
    div.appendChild(h1);
}
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <p>段落文本</p>
</div>
</body>
</html>
```

`insertBefore()` 方法的用法比较特殊，它可以把一个指定的节点插入到给定元素中。这与 `appendChild()` 方法略有不同，`insertBefore()` 方法所插入的节点位于指定元素的指定子节点的前面，而不是放置在最后面。因此该方法可以包含两个参数，具体用法如下：

```
var o = e.insertBefore(new_e, target_node);
```

返回值 `o` 表示新添加的子节点，参数 `new_e` 表示新添加的子节点。而参数 `target_node` 表示元素 `e` 包含的一个子节点。如果指定了 `target_node` 参数，则表示在 `target_node` 子节点前面增加一个指定子节点对象；如果 `target_node` 参数值为 `null`，则插入的子节点被放置在最后面，此时它等同于 `appendChild()` 方法。

【示例 3】 使用 `insertBefore()` 方法可以把指定元素精确插入到文档结构中任意位置。下面示例中在 `div` 元素的第一个子元素前面插入一个 `h1` 元素，动态插入的结构如图 4.4 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var h1 = document.createElement("h1");
    var o = div.insertBefore(h1,div.firstChild);
}
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <p>段落文本</p>
</div>
</body>
</html>

```



图 4.3 appendChild()方法的应用



图 4.4 insertBefore()方法的应用

jQuery 定义了 4 种方法用来在节点内部插入内容，说明如表 4.1 所示。

表 4.1 在节点内部插入内容的方法

方 法	说 明
append()	向每个匹配的元素内部追加内容
appendTo()	把所有匹配的元素追加到另一个指定的元素集合中。实际上，该方法颠倒了 append() 的用法。例如，\$(A).append(B) 与 \$(B).appendTo(A) 是等价的
prepend()	向每个匹配的元素内部前置内容
prependTo()	把所有匹配的元素前置到另一个指定的元素集合中。实际上，该方法颠倒了 prepend() 的用法。例如，\$(A).prepend(B) 与 \$(B).prependTo(A) 是等价的

1. append()方法

append()方法能够把参数指定的内容插入到指定的节点中，并返回一个 jQuery 对象。指定的内容被插入到每个匹配元素里面的最后面，作为它的最后一个子元素（last child）。具体用法如下：

```

append(content)
append(function(index,html))

```

参数 content 可以是一个元素、HTML 字符串或 jQuery 对象，用来插在每个匹配元素里的末尾。参数 function(index, html) 是一个返回 HTML 字符串的函数，该字符串用来插入到匹配元素的末尾。

【示例 4】调用 jQuery 的 append() 方法把一个列表项字符串添加到当前列表的末尾。演示效果如图 4.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".container").append('');
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>图标列表</h2>
<ul class="container">
    <li></li>
    <li></li>
</ul>
</body>
</html>
```

【提示】

append() 方法不仅接收 HTML 字符串，还可以接收 jQuery 对象和 DOM 对象。如果把 jQuery 对象追加到当前元素尾部，则将删除原来位置的 jQuery 匹配对象，此操作相当于移动，而不是复制。例如，下面用法将把标题移动到列表结构的尾部，显示效果如图 4.6 所示。

```
$(function(){
    $(".container").append($(".h2"));
})
```



图 4.5 在列表项末尾添加新项

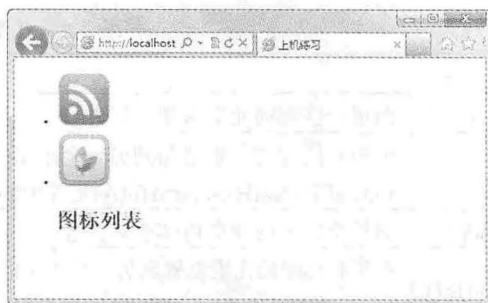


图 4.6 在文档中移动元素位置

2. appendTo() 方法

appendTo() 方法将匹配的元素插入到目标元素的最后面（里面的后面）。具体用法如下：

appendTo(target)

参数 target 表示一个选择符、元素、HTML 字符串或者 jQuery 对象；符合的元素会被插入到由参数指

定的目标末尾。例如，对于下面一行语句：

```
$(".container").append($("#h2"));
```

可以改写为：

```
$("#h2").appendTo($(".container"));
```

appendTo()方法与 append()方法功能相同，主要是语法不同，即内容和目标的位置不同。对于 append()方法来说，选择表达式在函数的前面，参数是将要插入的内容。而 appendTo()方法刚好相反，内容在方法前面，它将被放在参数里元素的末尾。

3. prepend()方法

prepend()方法能够把参数指定的内容插入到指定的节点中，并返回一个 jQuery 对象。指定的内容被插入到每个匹配元素里面的最前面，作为它的第 1 个子元素 (first child)。具体用法如下：

```
prepend(content)
```

```
prepend(function(index,html))
```

参数 content 可以是一个元素、HTML 字符串或 jQuery 对象，用来插在每个匹配元素里的末尾。参数 function(index, html)是一个返回 HTML 字符串的函数，该字符串用来插入到匹配元素的末尾。

【示例 5】调用 jQuery 的 prepend()方法把一个列表项字符串添加到当前列表的首位，演示效果如图 4.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $(".container").prepend('');
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>图标列表</h2>
<ul class="container">
    <li></li>
    <li></li>
</ul>
</body>
</html>
```



图 4.7 在列表项首位添加新项

其他用法与 `append()` 方法相同, 另外, jQuery 还定义了 `prependTo()` 方法, 该方法与 `appendTo()` 方法相对应, 即把指定的 jQuery 对象包含的内容插入到参数匹配的元素中。

4.2.2 外部插入

DOM 没有直接提供外部插入的一般方法, 如果要实现在匹配元素外面插入或者包裹元素, 则需要间接方式实现。但 jQuery 在这方面提供了很多实现外部插入内容的一般方法, 详细说明如表 4.2 所示。

表 4.2 在节点外部插入内容

方 法	说 明
<code>after()</code>	在每个匹配的元素之后插入内容
<code>before()</code>	在每个匹配的元素之前插入内容
<code>insertAfter()</code>	把所有匹配的元素插入到另一个指定的元素集合的后面
<code>insertBefore()</code>	把所有匹配的元素插入到另一个指定的元素集合的前面

1. `after()` 方法

`after()` 方法能够根据参数设定在每一个匹配的元素之后插入内容。具体用法如下:

```
after(content)
after(function(index))
```

参数 `content` 表示一个元素、HTML 字符串或 jQuery 对象, 用来插在每个匹配元素的后面。参数 `function(index)` 表示一个返回 HTML 字符串的函数, 这个字符串会被插入到每个匹配元素的后面。

【示例 6】调用 jQuery 的 `after()` 方法在每个列表项后面添加一行字符串, 该字符串是通过 `$("#li img").attr("src")` 方法从列表结构中获取第 1 个项目图标中的 `src` 属性值。演示效果如图 4.8 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#li img").after($("#li img").attr("src"));
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>图标列表</h2>
<ul class="container">
    <li></li>
    <li></li>
</ul>
</body>
</html>
```

2. `insertAfter()` 方法

`insertAfter()` 与 `after()` 方法具有相同的功能, 主要是语法不同, 特别是内容和目标的位置。对于 `after()` 方法来说, 选择表达式在函数的前面, 参数是要插入的内容。而 `insertAfter()` 方法刚好相反, 内容在方法前

面，它将被放在参数里元素的后面。具体用法如下：

`insertAfter(target)`



图 4.8 在列表项后面添加注释行文本

参数 `target` 表示一个选择器、元素、HTML 字符串或 jQuery 对象，匹配的元素将会被插入在由参数指定的目标后面。例如，针对下面这行代码：

```
$("#li img").after($("#<span>注释文本</span>"));
```

可以改写为：

```
$("#<span>注释文本</span>").insertAfter($("#li img"));
```

3. `before()`方法

`before()`方法将在每个匹配的元素之前插入内容。具体用法如下：

`before (content)`

`before (function(index))`

参数 `content` 表示一个元素、HTML 字符串或 jQuery 对象，用来插在每个匹配元素的后面。参数 `function(index)`表示一个返回 HTML 字符串的函数，这个字符串会被插入到每个匹配元素的后面。

【示例 7】调用 jQuery 的 `before()`方法在每个列表项前面添加一个 `span` 元素，并在该元素中显示提示性的字符串。演示效果如图 4.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#li img").before($("#li img").attr("src"));
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>图标列表</h2>
<ul class="container">
    <li></li>
    <li></li>
</ul>
</body>
</html>
```




图 4.9 在列表项前面添加注释行文本

4. insertBefore()方法

`insertBefore()`与 `before()`方法具有相同的功能，主要是语法不同，特别是内容和目标的位置。对于 `before()`方法来说，选择表达式在函数的前面，参数是将要插入的内容。而 `insertBefore()`方法刚好相反，内容在方法前面，它将被放在参数里元素的后面。具体用法如下：

```
insertBefore(target)
```

参数 `target` 表示一个选择器、元素、HTML 字符串或 jQuery 对象，匹配的元素将会被插入在由参数指定的目标后面。例如，针对下面这行代码：

```
$("#li img").before($("#<span>注释文本</span>"));
```

可以改写为：

```
$("#<span>注释文本</span>").before($("#li img"));
```

【提示】

`appendTo()`、`prependTo()`、`insertBefore()`和 `insertAfter()`方法具有破坏性操作特性。也就是说，如果选择已存在内容，并把它们插入到指定对象中时，则原位置的内容将被删除。

4.3 删除内容

DOM 内置了 `removeChild()`方法，该方法可以删除指定的节点及其包含的所有子节点，并返回这些删除的内容。具体用法如下：

```
nodeObject.removeChild(node)
```

其中，`nodeObject` 表示节点对象，参数 `node` 表示要删除的子节点。

【示例 8】 先使用 `document.getElementsByTagName()`方法获取页面中的 `div` 和 `p` 元素，然后移出 `p` 元素，把移出的 `p` 元素附加到 `div` 元素后面。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var p = document.getElementsByTagName("p")[0];
    var p1 = div.removeChild(p);
    div.parentNode.insertBefore(p1,div.nextSibling);
}
```

```

</script>
<title>上机练习</title>
</head>
<body>
<div>
  <p>段落文本</p>
</div>
</body>
</html>

```

由于 DOM 的 `insertBefore()` 与 `appendChild()` 方法都具有破坏性, 当使用文档中现有元素进行操作时, 会先删除原位置上的元素。因此对于下面两行代码:

```

var p1 = div.removeChild(p);           //移出 p 元素
div.parentNode.insertBefore(p1,div.nextSibling); //把移出的 p 元素附加到 div 元素后面

```

可以合并为:

```

div.parentNode.insertBefore(p,div.nextSibling); //直接使用 insertBefore()移动 p 元素

```

jQuery 定义了 3 种删除内容的方法: `remove()`、`empty()` 和 `detach()`。其中, `remove()` 方法对应 DOM 的 `removeChild()` 方法, 详细说明如表 4.3 所示。

表 4.3 jQuery 删除内容的方法

方 法	说 明
<code>remove()</code>	从 DOM 中删除所有匹配的元素
<code>empty()</code>	删除匹配的元素集合中所有的子节点
<code>detach()</code>	从 DOM 中删除所有匹配的元素

4.3.1 移出

`remove()` 方法能够将匹配元素从 DOM 中删除。该方法的用法如下:

```
remove([selector])
```

参数 `selector` 表示一个选择表达式, 用来过滤匹配的将被移除的元素。该方法还将同时移除元素内部的一切, 包括绑定的事件及与该元素相关的 jQuery 数据。

【示例 9】 为 `<button>` 标签绑定 `click` 事件, 当用户单击按钮时将调用 jQuery 的 `remove()` 方法移出所有的段落文本。演示效果如图 4.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
  $("button").click(function () {
    $("p").remove();
  });
})
</script>
<title>上机练习</title>
</head>
<body>

```

```

<p>段落文本 1</p>
<div>布局文本</div>
<p>段落文本 2</p>
<button>清除段落文本</button>
</body>
</html>

```

【提示】

由于 remove() 方法能够删除匹配的元素，并返回这个被删除的元素，因此在特定条件下该方法的功能可以使用 jQuery 的 appendTo()、prependTo()、insertBefore() 或 insertAfter() 方法进行模拟。例如，下面示例将父元素 div 的子元素 p 移出，然后插入到父元素 div 的后面。执行之后的 HTML 结构如图 4.11 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var $p = $("p").remove();
    $p.insertAfter("div");
})
</script>
<title>上机练习</title>
</head>
<body>
<div>
    <p>段落文本</p>
</div>
</body>
</html>

```



图 4.10 单击移出段落文本

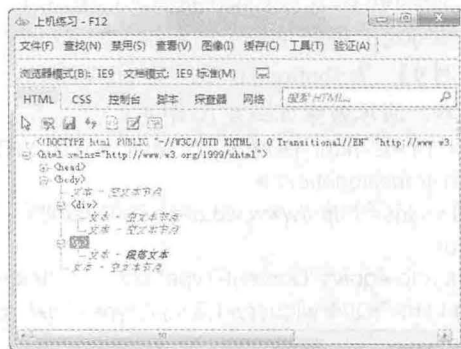


图 4.11 使用 jQuery 移动 HTML 结构

如果使用 insertAfter() 方法，则可以把上面的两步操作合并为一步，代码如下：

```

<script type="text/javascript">
$(function(){
    $("p").insertAfter("div"); //直接把段落文本移动到 div 元素
})
</script>

```

当然，remove() 方法的主要功能是删除指定节点以及包含的子节点。

4.3.2 清空

`empty()`方法不是删除节点，而是清空元素包含的内容。在用法上，`empty()`和 `remove()`方法相似，但是执行结果略有区别。该方法的用法如下：

`empty()`

该方法没有参数，表示将直接删除匹配元素包含的所有内容。

【示例 10】 为<button>标签绑定 click 事件，当用户单击按钮时将调用 jQuery 的 `empty()`方法移出段落文本内所有的内容，但没有删除 p 元素。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").click(function () {
        $("p").empty ();
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<div>布局文本</div>
<p>段落文本 2</p>
<button>清除段落文本</button>
</body>
</html>
```

【提示】

移出和清空是两个不同的操作概念。移出将删除指定的 jQuery 对象所匹配的所有元素，以及其包含的所有内容；而清空仅删除指定的 jQuery 对象所匹配的所有元素包含的内容，但是不删除当前匹配元素。

同时，`remove()`方法能够根据传递的参数进行有选择地移出操作，而 `empty()`方法将对所有匹配的元素执行清空操作，没有可以选择的参数。

4.3.3 分离

`detach()`方法能够将匹配元素从 DOM 中分离出来。具体用法如下：

`detach([expr])`

参数 `expr` 是一个选择表达式，将需要移出的元素从匹配的元素中过滤出来。该参数可以省略，如果省略将移出所有匹配的元素。

【示例 11】 为<button>标签绑定 click 事件，当用户单击按钮时将调用 jQuery 的 `detach()`方法移出所有的段落文本，演示效果如图 4.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){
        $(this).toggleClass("off");
    });
    var p;
    $("button").click(function(){
        if ( p ) {
            p.appendTo("body");
            p = null;
        } else {
            p = $("p").detach();
        }
    });
})
</script>
<style>
p {
    background:yellow;
    margin:6px 0;
    cursor:pointer;
}
p.off {background: white;}
</style>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<div>布局文本</div>
<p>段落文本 2</p>
<button>清除段落文本</button>
</body>
</html>

```

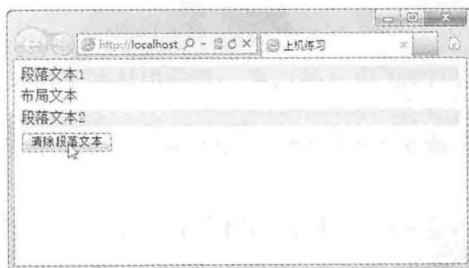


图 4.12 分离段落文本

【代码详解】

在示例 11 中, 文档中包含两段文本, 通过 `$("p").click()` 函数为段落文本绑定一个单击事件, 即单击段落文本时, 将设置或者移出样式类 `off`, 这样 `p` 元素就拥有了一个事件属性, 单击段落文本可以切换 `off` 样式类。在内部样式表中, 定义段落文本默认背景色为浅黄色, 单击后应用 `off` 样式类, 恢复默认的白色背景, 通过 `toggleClass()` 类切换方法实现再次单击段落文本后将再次显示浅黄色背景。

然后在按钮的 `click` 事件处理函数中, 将根据一个临时变量 `p` 的值来判断是否分离文档中的段落文本, 或者把分离的段落文本重新附加到文档尾部。此时, 读者会发现当再次恢复被删除的段落文本后, 它依然保留着上面定义的事件属性。

【注意】

`detach()` 方法与 `remove()` 方法一样, 其差别在于, `detach()` 方法能够保存所有 jQuery 数据与被移走的元素相关联, 所有绑定在元素上的事件、附加的数据等都会保留下来。当需要移走一个元素, 不久又将该元素插入 DOM 时, 这种方法很有用。

例如, 以示例 11 为例, 如果使用 `remove()` 方法代替 `detach()` 方法, 则当再次恢复被删除的段落文本后, 段落文本的 `click` 事件属性将失效。主要代码如下:

```
$(function(){
    $("p").click(function(){
        $(this).toggleClass("off");
    });
    var p;
    $("button").click(function(){
        if ( p ){
            p.appendTo("body");
            p = null;
        } else {
            p = $("p").remove();
        }
    });
});
```

4.4 克隆内容

DOM 定义了 `cloneNode()` 方法, 该方法可以创建指定节点的副本。具体用法如下:

```
nodeObject.cloneNode(include_all)
```

其中, `nodeObject` 表示节点对象, 一般为元素。`cloneNode()` 方法包含一个参数, 取值为 `true` 或 `false`, 用来设置被复制的节点是否包括原节点的所有属性和子节点。该方法返回被克隆的节点。

【示例 12】 使用 `cloneNode()` 方法复制 `div` 元素及其所有属性和子节点, 然后当单击段落文本时, 将复制段落文本, 并追加到文档的尾部。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
```



```

div.onclick = function(){
    var div1 = div.cloneNode(true);
    div.parentNode.insertBefore(div1,div.nextSibling);
}
}
</script>
<title>上机练习</title>
</head>
<body>
<div class="red" title="no" onclick="alert('ok')">
    <p>段落文本</p>
</div>
</body>
</html>

```

注意，复制的 div 元素不拥有鼠标单击事件，而拥有 div 元素自身携带的鼠标双击事件。如果为 clone() 方法传递 true 参数，则可以使复制的 div 元素也拥有单击事件。也就是说，当单击复制的 div 元素时，会继续进行复制操作，连续单击会使复制的 div 元素成倍增加。

jQuery 定义了 clone() 方法用来复制节点，jQuery 的 clone() 方法能够复制匹配的 DOM 元素并且选中这些复制的副本。具体用法如下：

```

clone( [ withDataAndEvents ] )
clone( [ withDataAndEvents ], [ deepWithDataAndEvents ] )

```

参数 withDataAndEvents 表示一个 Boolean 值（true 或 false），用来设置是否复制事件处理函数等数据。对于 jQuery 1.4 版本来说，元素数据也会被复制，默认值是 false。

参数 deepWithDataAndEvents 也是一个布尔值，用来设置是否对事件处理函数和克隆的元素的所有子元素的数据进行复制。默认情况下，它的值与第 1 个参数的值相匹配（默认值是 false）。

【示例 13】 通过 clone(true) 方法复制 标签，并把它复制到 <p> 标签的后面，同时保留该标签默认的事件处理函数。演示效果如图 4.13 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("b").click(function(){
        $(this).toggleClass("off");
    });
    $("b").clone(true).insertAfter("p");
})
</script>
<style>
.off {background: yellow;}
</style>
<title>上机练习</title>
</head>
<body>
<b>加粗文本</b>
<p>段落文本</p>

```

```

</body>
</html>

```



图 4.13 克隆内容

4.5 替换内容

DOM 定义了 `replaceChild()` 方法实现节点替换。具体用法如下：

```
nodeObject.replaceChild(new_node,old_node)
```

其中，`nodeObject` 表示节点对象。该方法包含两个参数，第 1 个参数指定替换节点，第 2 个参数指定被替换的节点。如替换成功，此方法可返回被替换的节点。如替换失败，则返回 `null`。

【示例 14】 使用 `document.createElement("div")` 方法创建一个 `div` 元素，然后在循环结构体内逐一使用克隆的 `div` 元素替换掉段落文本内容。演示效果如图 4.14 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var p = document.getElementsByTagName("p");
    var div = document.createElement("div");
    div.innerHTML = "盒子";
    for(var i=0,l = p.length;i<l;i++){
        var div1 = div.cloneNode(true);
        p[i].parentNode.replaceChild(div1,p[i]);
    }
}
</script>
<title>上机练习</title>
</head>
<body>
<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
</body>
</html>

```

jQuery 定义了 `replaceWith()` 和 `replaceAll()` 方法用来替换节点。与之对应，DOM 定义了 `replaceChild()` 方法实现相同的功能，不过它们的用法迥然不同。

1. `replaceWith()` 方法

`replaceWith()` 方法能够将所有匹配的元素替换成指定的 HTML 或 DOM 元素。具体用法如下：

```
replaceWith( newContent )
replaceWith( function )
```

参数 newContent 表示用来插入的内容，可以是 HTML 字符串、DOM 元素或 jQuery 对象。

参数 function 返回 HTML 字符串，表示用来替换的内容。

【示例 15】为按钮绑定 click 事件处理函数，当单击按钮后将调用 replaceWith() 方法把当前按钮替换为 div 元素，并把按钮显示的文本装入到 div 元素中。演示效果如图 4.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").click(function () {
        $(this).replaceWith("<div>" + $(this).text() + "</div>");
    });
})
</script>
<style type="text/css">
button {display:block; margin:3px; color:red; width:200px;}
div {color:red; border:2px solid blue; width:200px; margin:3px; text-align:center;}
</style>
<title>上机练习</title>
</head>
<body>
<button>按钮 1</button>
<button>按钮 2</button>
<button>按钮 3</button>
</body>
</html>
```

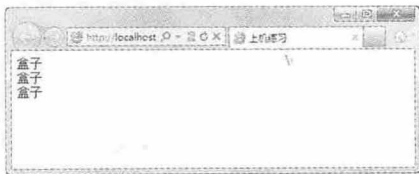


图 4.14 替换段落文本节点

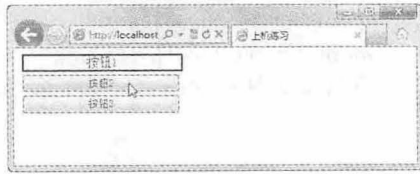


图 4.15 替换按钮

【提示】

replaceWith() 方法将会用选中的元素替换目标元素，此操作是移动，而不是复制。与大部分其他 jQuery 方法一样，replaceWith() 方法返回 jQuery 对象，所以可以通过链式语法与其他方法链接使用。但是需要注意的是，replaceWith() 方法返回的 jQuery 对象与被移走的元素相关联，而不是新插入的元素。在 jQuery 1.4 中，replaceWith()、before() 和 after() 方法都对分离的 DOM 元素有效。

2. replaceAll() 方法

replaceAll() 方法能够用匹配的元素替换掉所有指定参数匹配到的元素。具体用法如下：

```
replaceAll(selector)
```

参数 selector 表示 jQuery 选择器字符串，用于查找所要被替换的元素。

在 jQuery 1.3.2 版本中, `appendTo()`、`prependTo()`、`insertBefore()`、`insertAfter()`和 `replaceAll()`这几个方法都为破坏性操作, 要选择先前选中的元素, 需要使用 `end()`方法恢复原来的 jQuery 对象。

【示例 16】 `replaceAll()`方法与 `replaceWith()`方法实际是一对相反操作, 它们实现的结果是一致的, 但是行为方式相反, 类似 `$A.replaceAll($B)`等于 `$B.replaceWith($A)`。在下面示例中, 使用 `replaceAll()`方法替换示例 15 中的 `replacecWith()`方法, 所实现的结果都是一样的。即为按钮绑定 `click` 事件处理函数, 当单击按钮后将调用 `replaceAll()`方法把当前按钮替换为 `div` 元素, 并把按钮显示的文本装入到 `div` 元素中。演示效果如图 4.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("button").click(function () {
        $("<div>" + $(this).text() + "</div>").replaceAll(this);
    });
})
</script>
<style type="text/css">
button {display:block; margin:3px; color:red; width:200px;}
div {color:red; border:2px solid blue; width:200px; margin:3px; text-align:center;}
</style>
<title>上机练习</title>
</head>
<body>
<button>按钮 1</button>
<button>按钮 2</button>
<button>按钮 3</button>
</body>
</html>
```

4.6 包裹内容

在开发中经常需要将某个结构元素进行包裹。当需要在结构中插入额外的标记时, 这种包裹操作不会破坏文档原有的结构和语义。DOM 没有提供包裹元素的操作方法, jQuery 定义了 3 种包裹元素的方法: `wrap()`、`wrapAll()`和 `wrapInner()`。这些方法的主要区别就在于包裹的形式不同, 下面分别进行详细讲解。

4.6.1 外包

`wrap()`方法能够在每个匹配的元素外层包上一个 html 元素。具体用法如下:

```
wrap(wrappingElement)
wrap(wrappingFunction)
```

- ☑ 参数 `wrappingElement` 表示一个 HTML 片段、选择表达式、jQuery 对象或 DOM 元素, 用来包在匹配元素的外层。

☑ 参数 `wrappingFunction` 表示一个用来包元素的回调函数。

【示例 17】为每个匹配的 `<a>` 标签使用 `wrap()` 方法包裹一个 `` 标签，为了方便观察，在文档头部定义一个内部样式表，定义 `li` 元素显示红色边框样式。演示效果如图 4.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("a").wrap("<li></li>");
})
</script>
<style type="text/css">
li{border:solid 1px red; padding:2px;}
a{background:#FCF;}
</style>
<title>上机练习</title>
</head>
<body>
<a href="#">首页</a>
<a href="#">社区</a>
<a href="#">新闻</a>
</body>
</html>
```

【提示】

参数可以是字符串或者对象，只要该参数能够形成 DOM 结构即可，且 jQuery 允许参数嵌套，但是结构只包含一个最里层元素，这个结构会包在每个匹配元素外层。该方法返回没被包裹过的元素的 jQuery 对象，用来链接其他函数。例如，针对示例 17，把其中的代码行：

```
$("a").wrap("<li></li>");
```

替换为：

```
$("a").wrap("<ul><li></li></ul>");
```

然后在内部样式表中添加 `ul{border:solid 2px blue;}` 样式，在浏览器中预览演示效果，如图 4.17 所示。

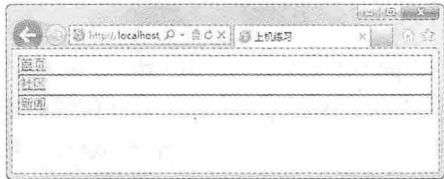


图 4.16 为超链接包裹项目列表

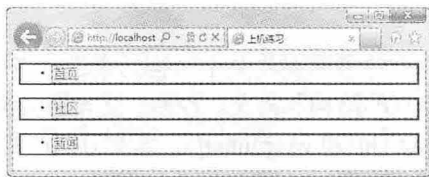


图 4.17 为超链接包裹多层的结构

4.6.2 内包

`wrapInner()` 方法能够为匹配元素里的内容外包一层结构。具体用法如下：

```
wrapInner( wrappingElement )
```

```
wrapInner ( wrappingFunction )
```

☑ 参数 `wrappingElement` 表示一个 HTML 片段、选择表达式、jQuery 对象或 DOM 元素，用来包在匹

配元素内的内容外层。

☑ 参数 `wrappingFunction` 表示一个用来包元素的回调函数。

【示例 18】先为每个匹配的 `<a>` 标签使用 `wrap()` 方法包裹一个 `` 标签，然后在 `body` 元素内使用 `wrapInner()` 方法为所有列表项包裹一个 `ul` 元素。为了方便观察，在文档头部定义一个内部样式表，定义 `li` 元素显示红色边框样式，同时定义 `ul` 元素显示为蓝色粗边框线。演示效果如图 4.18 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("a").wrap("<li></li>");
    $("body").wrapInner("<ul></ul>");
})
</script>
<style type="text/css">
ul{border:solid 2px blue;}
li{border:solid 1px red; padding:2px;}
a{background:#FCF;}
</style>
<title>上机练习</title>
</head>
<body>
<a href="#">首页</a>
<a href="#">社区</a>
<a href="#">新闻</a>
</body>
</html>
```

【提示】

与 `wrap()` 方法一样，参数可以是字符串或者对象，只要该参数能够形成 DOM 结构即可，且 jQuery 允许参数嵌套，但是结构只包含一个最里层元素，这个结构会包在每个匹配元素外层。该方法返回没被包裹过的元素的 jQuery 对象，用来链接其他函数。例如，针对示例 18，把其中的代码行：

```
$("body").wrapInner("<ul></ul>");
```

替换为：

```
$("body").wrapInner("<div><div><ul></ul></div></div>");
```

然后在内部样式表中添加 `div{border:solid 1px gray; padding:5px;}` 样式，在浏览器中预览演示效果，如图 4.19 所示。

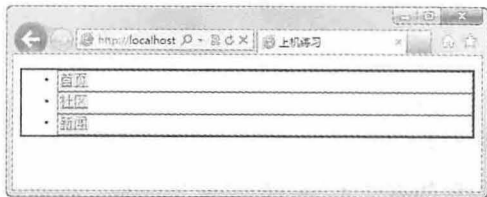


图 4.18 为网页内容包裹列表框

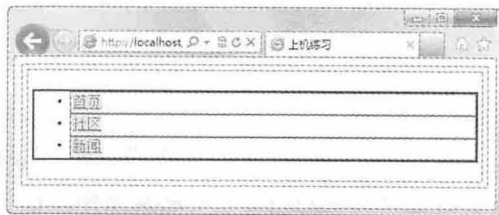


图 4.19 为网页内容包裹多层的结构

4.6.3 总包

wrapAll()方法能够在所有匹配元素外包一层结构。具体用法如下：

wrapAll(wrappingElement)

参数 wrappingElement 表示用来包在外面的 HTML 片段、选择表达式、jQuery 对象或者 DOM 元素。

【示例 19】 先为每个匹配的<a>标签使用 wrap()方法包裹一个标签，然后使用 wrapAll()方法为所有列表项包裹一个 ul 元素。为了方便观察，在文档头部定义一个内部样式表，定义 li 元素显示红色边框样式，同时定义 ul 元素显示为蓝色粗边框线。动态结构图如图 4.20 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("a").wrap("<li></li>");
    $("li").wrapAll("<ul></ul>");
})
</script>
<style type="text/css">
ul{border:solid 2px blue;}
li{border:solid 1px red; padding:2px;}
a{background:#FCF;}
</style>
<title>上机练习</title>
</head>
<body>
<a href="#">首页</a>
<a href="#">社区</a>
<a href="#">新闻</a>
</body>
</html>
```



图 4.20 为列表项包裹一个列表结构

本示例演示效果与 4.6.2 节的效果图一样，虽然它们使用的方法不同，但是结果一致。也就是说，\$("li").wrapAll("")等效于\$("body").wrapInner("")，当然要确保它们结构的一致性。

4.6.4 卸包

unwrap()方法能够将匹配元素的父级元素删除，保留自身（和兄弟元素，如果存在）在原来的位置。与wrap()方法的功能相反。该方法没有参数，具体用法如下：

unwrap()

【示例 20】为按钮绑定一个开关事件，这样单击按钮时可以为<a>标签包裹或者卸包标签。在浏览器中的预览效果如图 4.21 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").toggle(function(){
        $("a").wrap("<li></li>");
    }, function(){
        $("a").unwrap();
    });
})
</script>
<style type="text/css">
li{border:solid 1px red; padding:2px;}
a{background:#FCF;}
</style>
<title>上机练习</title>
</head>
<body>
<button>包装/卸包</button>
<a href="#">首页</a>
<a href="#">社区</a>
<a href="#">新闻</a>
</body>
</html>
```



图 4.21 包裹或者卸包<a>标签

4.7 属性操作

属性操作包括设置元素的属性、读取元素属性、删除属性或者修改属性值等。jQuery 和 DOM 都提供了

元素属性的基本操作方法。掌握这些方法可以满足 Web 开发中的普通应用需求，下面分别进行讲解。

4.7.1 设置属性

在 DOM 中使用 `setAttribute()` 方法可以设置元素属性，具体用法如下：

`elementNode.setAttribute(name,value)`

其中 `elementNode` 表示元素节点，参数 `name` 表示设置的属性名，`value` 表示要设置的属性值。

【示例 21】为页面中段落文本标签 `<p>` 定义一个 `title` 属性，设置属性值为“段落文本”。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.setAttribute("title","段落文本");
}
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>
```

jQuery 定义了两个用来设置元素属性值的方法：`prop()` 和 `attr()`，下面分别进行介绍。

1. `prop()` 方法

`prop()` 能够为匹配的元素设置一个或更多的属性。具体用法如下：

`prop(propertyName, value)`

`prop(map)`

`prop(propertyName, function(index, oldPropertyValue))`

☑ 参数 `propertyName` 表示要设置的属性的名称，`value` 表示一个值，用来设置属性值。如果为元素设置多个属性值，可以使用 `map` 参数，该参数是一个用于设置属性的对象，以 {属性:值} 对形式进行定义。

☑ 参数 `function(index,oldPropertyValue)` 用来设置返回值的函数。接收到集合中的元素和属性的值作为参数旧的索引位置。在函数中，关键字 `this` 指的是当前元素。

【示例 22】先为所有被勾选的复选框设置只读属性，即当 `input` 元素的 `checked` 属性值为 `checked` 时，则调用 `prop()` 方法设置该元素的 `disabled` 属性值为 `true`。在浏览器中的预览效果如图 4.22 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("input[checked='checked']").prop({
        disabled: true
```



```

});
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="checkbox" checked="checked" />
<input type="checkbox" />
<input type="checkbox" />
<input type="checkbox" checked="checked" />
</body>
</html>

```

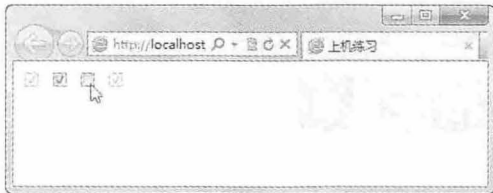


图 4.22 为复选框设置只读属性

【提示】

prop()方法常用来方便地设置属性的函数值，尤其是设置多个属性或通过使用返回值时。一般在不影响性能的情况下改变属性的序列化的 HTML DOM 元素的动态，如包括 value 属性的 input 元素、disabled 属性的 inputs 和按钮以及 checked 属性的复选框。大多数情况下，prop()应该用于设置 disabled 和 checked 属性值，而不应该使用 attr()方法。此时建议使用 val()方法获取值。

2. attr()方法

attr()也能够为匹配的元素设置一个或更多的属性。具体用法如下：

attr(attributeName, value)

attr(map)

attr(attributeName, function(index, attr))

- ☑ 参数 attributeName 表示要设置的属性的名称，value 表示一个值，用来设置属性值。如果为元素设置多个属性值，可以使用 map 参数，该参数是一个用于设置属性的对象，以{属性:值}对形式进行定义。
- ☑ 参数 function(index, attr)为用来设置返回值的函数。接收到集合中的元素和属性的值作为参数旧的索引位置。在函数中，关键字 this 指的是当前元素。

attr()方法是设置属性值最方便、最强大的方法，特别是设置多个属性或使用值来自函数时。当设置多个属性，则包含属性名的引号是可选的。但是当设置样式名（class）属性时，必须使用引号。注意，IE 浏览器不允许使用该属性改变<input>或者<button>元素的 type 属性。

【示例 23】 使用 attr()方法为所有 img 元素动态设置 src 属性值，实现图像占位符自动显示序列图标图像效果。在浏览器中的预览效果如图 4.23 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">

```

```
$(function(){
    $("img").attr("src",function(index){
        return "images/icon ("+(index+1)+").png";
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<img /><img /><img />
</body>
</html>
```



图 4.23 动态设置 img 元素的 src 属性值

【提示】

attr()和 prop()方法都可以用来设置元素属性，但是它们在用法上还是有细微区别的。在 jQuery 1.6 版本之前全部使用 attr()方法来访问对象的属性，但是当使用它来访问 checkbox 的 disabled 属性时，会有些问题。在有些浏览器中，只要写了 disabled 就可以，有些则要写成 disabled = "disabled"。所以，从 jQuery 1.6 开始，jQuery 提供新的 prop()方法来获取这些属性。

使用 prop()方法时，返回值是标准属性，如\$("#checkbox").prop('disabled')，不会返回 disabled 或者空字符串，只会是 true/false。当然赋值时也是如此。如此无论是从语法上还是语义上便统一了所有操作。

哪些属性应该用 attr()方法访问，哪些应该用 prop()方法访问呢？注意以下两个原则即可：

- ☑ 只添加属性名，而不添加属性值就会生效。
- ☑ 只要属性值为 true/false 的属性。

详细说明如表 4.4 所示。

表 4.4 attr()和 prop()方法用法比较

Attribute/Property	attr()	prop()
accesskey	√	
align	√	
async		√
autofocus		√
checked		√
class	√	
contenteditable	√	
draggable	√	
href	√	
id	√	

续表

Attribute/Property	attr()	prop()
label	√	
location (IE window.location)		√
multiple		√
readOnly		√
rel	√	
selected		√
src	√	
tabindex	√	
title	√	
type	√	
width (if needed over width())	√	

4.7.2 访问属性

使用 DOM 定义的 `getAttribute()` 方法可以访问元素的属性，并获取属性的值。具体用法如下：

`elementNode.getAttribute(name)`

其中 `elementNode` 表示元素节点对象，参数 `name` 表示属性的名称，以字符串形式传递，该方法的返回值为指定属性的属性值。

【示例 24】 直接使用 JavaScript 读取段落文本中的 `title` 属性值，然后以提示对话框的形式显示出来。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    alert(p.getAttribute("title"));
}
</script>
<title>上机练习</title>
</head>
<body>
<p title="段落文本">段落文本</p>
</body>
</html>
```

与设置属性的方法一样，jQuery 定义了两个用来访问元素属性值的方法：`prop()`和`attr()`，这两个方法的用法在 4.7.1 节中曾经详细讲解。

当为 `prop()`和`attr()`方法传递两个参数时，一般用来为指定的属性设置值。而当为这两个方法传递一个参数时，则表示读取指定属性的值。

1. `prop()`方法

`prop()`方法的具体用法如下：

prop(propertyName)

参数 propertyName 表示要读取属性的名称。

【提示】

prop()方法只获得 jQuery 对象中第 1 个匹配元素的属性值。如果元素的一个属性没有设置,或者如果没有匹配的元素,则该方法将返回 undefined 值。为了得到每个元素单独的值,不妨使用循环结构的 jQuery.each() 或.map()方法来逐一读取。

attributes 和 properties 之间的差异在特定情况下是很重要的。在 jQuery 1.6 版本之前,attr()方法主要负责读取所有属性的值,有时检索时考虑到了一些属性的属性值,这可能导致不一致的行为。从 jQuery 1.6 版本开始, jQuery 新增加了 prop()方法,该方法提供了一种明确检索属性值,用来解决 attr()方法在检索属性时低效和返回值不一致的行为。

例如,针对下面这行 HTML 片段结构:

```
<input type="checkbox" checked="checked" />
```

假设在 JavaScript 中定义变量 elem 用来捕获该标签对象,则使用不同的方法访问该对象的 checked 属性时返回值是不同的:

```
$(elem).prop("checked")      //返回布尔值 true
elem.getAttribute("checked")  //返回字符串 checked
$(elem).attr("checked")(1.6+) //返回字符串 checked
$(elem).attr("checked")(pre-1.6) //返回布尔值 true
```

但是,根据 W3C 的表单规范,checked 属性是一个布尔属性,这意味着该属性值为布尔值。那么如果属性没有值或者为空字符串,这就为在脚本中进行逻辑判断带来了麻烦。考虑到不同浏览器对其处理结果不同,需要考虑一种跨浏览器兼容的方法,来确定一个复选框为被选中状态,为此可以采用下面方式之一进行检测:

```
if (elem.checked)
if ($(elem).prop("checked"))
if ($(elem).is(":checked"))
```

如果使用 attr()进行检测,就容易出现问題,因为 attr("checked")将获取该属性值,即只是用来存储默认或选中属性的初始值,无法直观地检测复选框的选中状态。因此使用下面代码检测复选框选中状态将是错误的:

```
if ( $(elem).attr("checked") )
```

【示例 25】 为复选框绑定 change()事件,当复选框状态发生变化后,将再次调用 change()方法,在该方法内通过参数函数动态获取当前复选框的状态值以及 checked 属性值,并分别使用 attr()、prop()和 is()方法来进行检测,以比较使用这 3 种方法所获取的值的差异。演示效果如图 4.24 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").change(function() {
        var $input = $(this);
        $("p").html("attr('checked') = <b>" + $input.attr('checked') + "</b><br>"
            + ".prop('checked') = <b>" + $input.prop('checked') + "</b><br>"
            + ".is(':checked') = <b>" + $input.is(':checked') + "</b>");
    }).change();
})
```

```

</script>
<style>
b {color: red;}
</style>
<title>上机练习</title>
</head>
<body>
<input id="check" type="checkbox" checked="checked">
<label for="check">复选框</label>
<p></p>
</body>
</html>

```



图 4.24 检测复选框的 checked 属性

2. attr()方法

attr()方法的具体用法如下:

attr(attributeName)

参数 attributeName 表示要读取属性的名称。

【提示】

与 prop()方法一样, attr()方法只获取 jQuery 第 1 个匹配元素的属性值。如果要获取每个单独的元素的属性值,需要用 jQuery 的 each()或者 map()方法做一个循环。

在 jQuery 1.6 版本中,当属性没有被设置时,attr()方法将返回 undefined。另外,attr()方法不应该用在普通的对象、数组、窗口(window)或文件(document)上。若要检索和更改 DOM 属性需使用 prop()方法。在开发中使用 jQuery 的 attr()方法来获取一个字符串值将有以下两个好处:

- ☑ 方便。可以直接被 jQuery 对象访问和链式调用其他 jQuery 方法。
- ☑ 兼容。一些属性在浏览器之间命名和取值不一致,甚至在同一个浏览器的不同版本中也存在差异。使用 attr()方法可以减少兼容性问题。

【示例 26】通过调用 jQuery 的 each()方法遍历所有匹配的 img 元素,然后在每个 img 元素的回调函数中分别使用 attr()方法获取该 img 元素的 title 属性值,并把它放在<p>标签中,然后把该段落文本追加到 img 元素的后面。演示效果如图 4.25 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("img").each(function(){
        $(this).after("<p>" + $(this).attr("title") + "</p>");
    });
});

```

```

    })
  })
</script>
<title>上机练习</title>
</head>
<body>



</body>
</html>

```

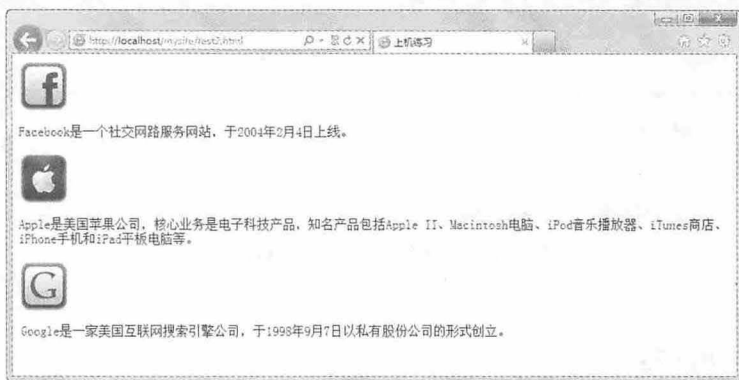


图 4.25 attr()方法在 jQuery 对象集合中的应用

4.7.3 删除属性

使用 DOM 定义的 removeAttribute()方法可以删除指定的元素属性。具体用法如下：

`elementNode.removeAttribute(name)`

其中 `elementNode` 表示元素节点对象，参数 `name` 表示属性的名称，以字符串形式传递。删除不存在的属性，或者删除没有设置但具有默认值属性时，删除操作将被忽略。如果文档类型声明（DTD）为指定的属性设置了默认值，那么再次调用 `getAttribute()`方法将返回那个默认值。

【示例 27】 直接使用 `removeAttribute()`方法删除段落文本中的 `title` 属性。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.removeAttribute("title");
}
</script>
<title>上机练习</title>
</head>
<body>
<p title="段落文本">段落文本</p>

```



```
</body>
</html>
```

jQuery 定义了 `removeProp()` 方法和 `removeAttr()` 方法都可以删除指定的元素属性，下面分别进行介绍。

1. `removeProp()` 方法

`removeProp()` 方法主要用来删除由 `prop()` 方法设置的属性集。对于一些内置属性的 DOM 元素或 Window 对象，如果试图删除部分属性，浏览器可能会产生错误。jQuery 第 1 次可能会错误地删除属性，分配给它一个 `undefined` 属性值，这样就避免了浏览器生成的任何错误。`removeProp()` 的具体用法如下：

```
removeProp(propertyName)
```

参数 `propertyName` 表示要删除的属性名称。

【示例 28】首先使用 `prop()` 方法为 `img` 元素添加一个 `code` 属性，然后访问该属性值，接着调用 `removeProp()` 方法删除 `code` 属性值，再次使用 `prop()` 方法访问属性，则显示值为 `undefined`。演示效果如图 4.26 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $img = $("img");
    $img.prop("code", 1234);
    $img.after("<div>图像密码初设置: " + String($img.prop("code")) + "</div>");
    $img.removeProp("code");
    $img.after("<div>图像密码现在是: " + String($img.prop("code")) + "</div>");
})
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>
```

2. `removeAttr()` 方法

`removeAttr()` 方法使用 DOM 原生的 `removeAttribute()` 方法，使用该方法的优点是能够直接被 jQuery 对象访问调用，而且具有良好的浏览器兼容性。对于特殊的属性，建议使用 `removeProp()` 方法。`removeAttr()` 方法的具体用法如下：

```
removeAttr(attributeName)
```

参数 `attributeName` 表示要删除的属性名称。

【示例 29】为按钮绑定 `click` 事件处理函数，当单击按钮时则调用 `removeAttr()` 方法移出文本框的 `disabled` 属性，再调用 `focus()` 方法激活文本框的焦点，并设置文本框的默认值为“可编辑文本框”。演示效果如图 4.27 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
```

```

<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("button").click(function () {
        $(this).next().removeAttr("disabled")
            .focus()
            .val("可编辑文本框");
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<button>激活文本框</button>
<input type="text" disabled="disabled" value="只读文本框" />
</body>
</html>

```

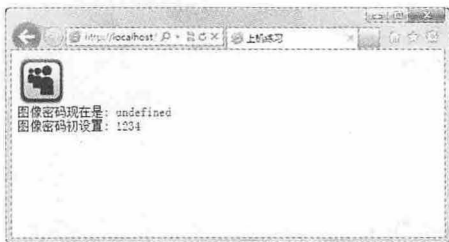


图 4.26 removeProp()方法的应用



图 4.27 removeAttr()方法的应用

4.8 类 操 作

在设计动态样式时,经常需要操作元素的 class 属性,该属性可以为元素定义类样式。作为元素的属性,可以使用 jQuery 的 attr()和 prop()方法或者 DOM 的 setAttribute()和 getAttribute()方法设置和读取元素的类样式。不过为了方便设计师操作, jQuery 单独定义了几个与类样式相关的操作方法。

4.8.1 添加类样式

DOM 没有定义专门的类操作方法,如果希望直接使用 JavaScript 控制类样式,则必须结合 setAttribute()和 getAttribute()方法来实现。

jQuery 定义了 addClass()方法专门负责为元素追加样式。具体用法如下:

```

addClass(className)
addClass(function(index,class))

```

- ☑ 参数 className 表示为每个匹配元素所要增加的一个或多个样式名。
- ☑ 参数函数 function(index, class)返回一个或多个用空格隔开的要增加的样式名,该参数函数够接收元素的索引位置和元素旧的样式名作为参数。

【示例 30】 使用 addClass()方法分别为文档中第 2、3 段添加不同的类样式,其中第 2 段添加类名 highlight,设计高亮背景显示,第 3 段添加类名 selected,设计文本加粗显示。演示效果如图 4.28 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

```



```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p:last").addClass("selected");
    $("p").eq(1).addClass("highlight");
})
</script>
<style>
.selected { font-weight:bold; }
.highlight { background:yellow; }
</style>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
</body>
</html>

```



图 4.28 addClass()方法的应用

【提示】

addClass()方法不会替换一个样式类名，它只是简单地添加一个样式类名到可能已经指定的元素上。对所有匹配的元素可以同时添加多个样式类名，样式类名通过空格分隔。例如：

```
$(p').addClass('class1 class2');
```

一般 addClass()方法与 removeClass()方法一起使用来切换元素的样式。例如：

```
$(p').removeClass('class1 class2').addClass(' class3');
```

从 jQuery 1.4 版本开始，.addClass()方法允许通过函数传递来设置样式名。例如：

```

$(ul li:last').addClass(function() {
    return 'item-' + $(this).index();
});

```

4.8.2 删除类样式

如果清空或者重设元素的 class 属性值，使用 jQuery 的 attr()和 prop()方法，或者 DOM 的 setAttribute()方法即可。但是如果要移出复合类样式中的某个样式，就需要另想办法。为了解决这个问题，jQuery 单独定义了 removeClass()方法，具体用法如下：

```

removeClass([className])
removeClass(function(index, class)

```


- ☑ 参数 `className` 为每个匹配元素移除的样式属性名。
- ☑ 参数函数 `function(index,class)` 返回一个或更多用空格隔开的被移除样式名, 该参数函数能够接收元素的索引位置和元素旧的样式名作为参数。

【示例 31】使用 `removeClass()` 方法分别删除偶数行段落文本的 `blue` 和 `under` 类样式。演示效果如图 4.29 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p:odd").removeClass("blue under");
})
</script>
<style>
.blue { color:blue; }
.under { text-decoration:underline; }
.highlight { background:yellow; }
</style>
<title>上机练习</title>
</head>
<body>
<p class="blue under">段落文本 1</p>
<p class="blue under highlight">段落文本 2</p>
<p class="blue under">段落文本 3</p>
<p class="blue under">段落文本 4</p>
</body>
</html>
```



图 4.29 `removeClass()` 方法的应用

【提示】

如果以一个样式类名作为参数, 那么只有该类所匹配的元素从集合中被删除; 如果没有样式名作为参数, 那么所有的样式类将被移除。从所有匹配的每个元素中同时移除多个用空格隔开的样式类, 例如:

```
$('#p').removeClass('class1 class2')
```

4.8.3 切换类样式

样式切换在 Web 开发中非常实用, 如折叠、开关、伸缩、Tab 切换等动态效果都需要用到交互切换。jQuery 定义了 `toggleClass()` 方法, 该方法可以开/关指定的类样式, 从而实现切换样式的设计目标。具体用法如下:

```
toggleClass(className)
toggleClass(className,switch)
toggleClass(function(index,class),[switch])
```

☑ 参数 `className` 表示在匹配的元素集合中的每个元素上用来切换的一个或多个（用空格隔开）样式类名。`switch` 表示一个用来判断样式类添加还是移除的 `boolean` 值。

☑ 参数函数 `function(index,class)` 用来返回在匹配的元素集合中的每个元素上用来切换的样式类名，该参数函数接收元素的索引位置和元素旧的样式类作为参数。

【示例 32】为文档中的按钮绑定 `click` 事件处理函数，当单击该按钮时将为 `p` 元素调用 `toggleClass()` 方法，并传递 `hidden` 类样式，实现段落包含的图像隐藏或者显示。演示效果如图 4.30 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").eq(0).click(function(){
        $("p").toggleClass("hidden");
    })
})
</script>
<style>
.hidden { display:none; }
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="切换样式" />
<p></p>
</body>
</html>
```

【提示】

`toggleClass()` 方法以一个或多个样式类名称作为参数。

如果在匹配的元素集合中的每个元素上存在该样式类就会被移除；如果某个元素没有这个样式类就会加上这个样式类。如果该方法包含第 2 个参数，则使用第 2 个参数判断样式类是否应该被添加或删除。如果这个参数的值是 `true`，那么这个样式类将被添加；如果这个参数的值是 `false`，那么这个样式类将被移除。

从 jQuery 1.4 版本开始，`toggleClass()` 方法允许通过函数来传递切换的样式类名。例如：

```
$("p").toggleClass(function() {
    if ($(this).parent().is('.bar')) {
        return 'happy';
    } else {
        return 'sad';
    }
});
```



图 4.30 toggleClass() 方法的应用

以代码表示如果匹配元素的父级元素有 bar 样式类名, 则为<p>元素切换 happy 样式类, 否则将切换 sad 样式类。

4.8.4 判断样式

DOM 定义了 hasAttribute() 方法, 使用该方法可以判断元素是否设置为指定属性。具体用法如下:

hasAttribute(name)

参数 name 表示属性名, 但是复合类样式中, 该方法无法判断 class 属性中是否包含了特定的类样式。为此, jQuery 定义了 hasClass() 方法用来判断元素是否包含指定的类样式。

.hasClass(className)

参数 className 表示要查询的样式名。例如:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    alert($("#p").hasClass("red"));    //返回 true
})

</script>
<title>上机练习</title>
</head>
<body>
<p class="red">段落文本</p>
</body>
</html>
```

hasClass() 方法实际上是 is() 方法的再包装, jQuery 为了方便用户使用, 重新定义了 hasClass() 专门用来判断指定类样式是否存在。其中, \$("#p").hasClass("red") 可以改写为 \$("#p").is(".red")。

4.9 读写文本和值

直接通过 DOM 结构树来操作文档, 有时候会感觉很麻烦, 而把 HTML 文档结构视为字符串, 并以字符串的形式进行操作, 会感觉很多问题迎刃而解, 就不用再考虑节点对象和节点关系了。另外, 对于文本节点来说, 直接把它视为字符串进行操作, 更符合一般人的思维习惯。

4.9.1 读写 HTML

DOM 为每个元素对象定义了 innerHTML 属性, 该属性以字符串形式读写元素包含的 HTML 结构。

【示例 33】使用 innerHTML 属性访问 div 元素包含的所有内容, 然后把这些内容通过 innerHTML 属性传递给 p 元素, 并覆盖掉 p 元素包含的文本。演示效果如图 4.31 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```



```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var div = document.getElementsByTagName("div")[0];
    p.innerHTML = div.innerHTML;
}
</script>
<style>
div { border:solid 2px red;}
p{ border:solid 1px blue;}
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <h1>标题</h1>
    <p>段落文本</p>
</div>
</body>
</html>

```

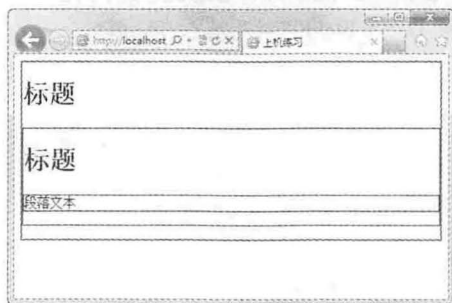


图 4.31 innerHTML 属性应用

jQuery 定义了 `html()` 方法，该方法可以以字符串形式读写 HTML 文档结构。具体用法如下：

```

html()
html(htmlString)
html(function(index,html))

```

- ☑ 参数 `htmlString` 用来设置每个匹配元素的一个 HTML 字符串。
- ☑ 参数函数 `function(index, html)` 用来返回设置 HTML 内容的一个函数，该参数函数接收元素的索引位置和元素旧的 HTML 作为参数。

当 `html()` 方法不包含参数时，表示以字符串形式读取指定节点下的所有 HTML 结构。当 `html()` 方法包含参数时，表示向指定节点下写入 HTML 结构字符串，同时会覆盖该节点原来包含的所有内容。

例如，针对示例 33 使用 jQuery 的 `html()` 方法实现的代码如下：

```

$(function(){
    var s = $("div").html();
    $("p").html(s);
})

```

注意，`html()` 方法实际上是对 DOM 的 `innerHTML` 属性包装，因此它不支持 XML 文档。

4.9.2 读写文本

DOM 中包含一个 `innerText` 属性, 该属性可以读写元素包含的文本。遗憾的是, `innerText` 属性存在兼容性问题, 很多标准浏览器不支持该属性。

jQuery 定义了 `text()` 方法, 使用该方法可以读写指定元素下包含的文本内容, 这些文本内容主要是指文本节点包含的数据。具体用法如下:

```
text(textString)
```

```
text(function(index,text))
```

☑ 参数 `textString` 用于设置匹配元素内容的文本。

☑ 参数函数 `function(index, text)` 用来返回设置文本内容的一个函数, 该参数函数可以接收元素的索引位置和元素旧的文本值作为参数。

当 `text()` 方法不包含参数时, 表示以字符串形式读取指定节点下的所有文本内容。当 `text()` 方法包含参数时, 表示向指定节点下写入文本字符串, 同时会覆盖该节点原来包含的所有文本内容。

【示例 34】 使用 `text()` 方法访问 `div` 元素包含的所有内容, 然后把这些内容通过 `text()` 方法传递给 `p` 元素, 并覆盖掉 `p` 元素包含的文本。演示效果如图 4.32 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var s = $("div").text();
    $("p").text(s);
})
</script>
<style>
div { border:solid 2px red;}
p{ border:solid 1px blue;}
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <h1>标题</h1>
    <p>段落文本</p>
</div>
</body>
</html>
```

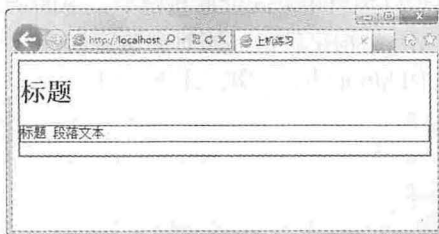


图 4.32 `text()` 方法的应用

4.9.3 读写值

值是一类特殊的文本字符串，主要是指表单元素中 `value` 属性设置的值。在 JavaScript 中可以通过元素对象的 `value` 属性来进行读写。不过对于下拉列表框、菜单等表单对象，在使用 `value` 属性直接读取时会存在很多兼容性问题。

jQuery 定义了 `val()` 方法用来读写指定表单元素包含的值。当 `val()` 方法不包含参数并调用时，表示将读取指定表单元素的值。当 `val()` 方法包含参数时，表示向指定表单元素写入值。具体用法如下：

```
val()
val(value)
val(function(index,value))
```

- ☑ 参数 `value` 表示一个文本字符串或一个以字符串形式的数组来设定每个匹配元素的值。
- ☑ 参数函数 `function(index,value)` 表示一个用来返回设置值的函数。

【示例 35】 下面示例演示了当文本框获取焦点时，清空默认的提示文本信息，准备用户输入值，而当离开文本框后，如果文本框没有输入信息，则重新显示默认的值。演示效果如图 4.33 所示。

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").focus(function(){
        if($(this).val() == "请输入文本") $(this).val("");
    })
    $("input").blur(function(){
        if($(this).val() == "") $(this).val("请输入文本");
    })
})
</script>
<title>上机练习</title>
</head>
<body>
<form action="" method="get">
    <input type="text" value="请输入文本" />
</form>
</body>
</html>
```

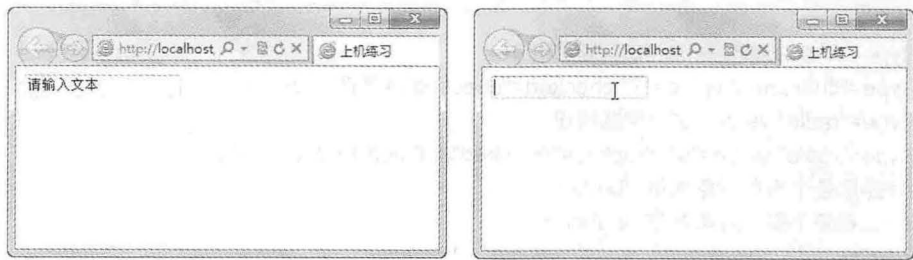


图 4.33 `val()` 方法的应用

【提示】

val()方法在读写单选按钮、复选框、下拉菜单和列表框的值时,比较实用且操作速度比较快。对于 val()方法来说,可以传递一个参数设置表单的显示值。由于下拉菜单和列表框显示为每个选项的文本,而不是 value 属性值,所以通过设置选项的显示值可以决定应显示的项目。不过对于其他表单元素来说,必须指定 value 属性值才有效。如果为元素指定多个值,则可以以数组的形式进行参数传递。

【示例 36】 单击第 1 个按钮可以使用 val()方法读取各个表单的值,单击第 2 个按钮可以设置表格表单的值。演示效果如图 4.34 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").eq(0).click(function(){
        alert($("#s1").val() + $("#s2").val() + $("input").val() + $(":radio").val());
    })
    $("button").eq(1).click(function(){
        $("#s1").val("单选 2");
        $("#s2").val(["多选 2", "多选 3"]);
        $("input").val(["6", "8"]);
    })
})
</script>
<title>上机练习</title>
</head>
<body>
<form action="" method="get">
    <select id="s1">
        <option value="1" selected="selected">单选 1</option>
        <option value="2">单选 2</option>
    </select>
    <select id="s2" size="3" multiple="multiple">
        <option value="3" selected="selected">多选 1</option>
        <option value="4">多选 2</option>
        <option value="5" selected="selected">多选 3</option>
    </select>
    <input type="checkbox" value="6"/>复选框 1
    <input type="checkbox" value="7" checked="checked"/>复选框 2<br />
    <input type="radio" value="8"/>单选按钮 1
    <input type="radio" value="9" checked="checked"/>单选按钮 2<br /><br />
    <button>显示各个表单对象的值</button>
    <button>设置各个表单对象的值</button>
</form>
</body>
</html>
```



图 4.34 val()方法的应用

4.10 样式表操作

借助 DOM 的 style 属性,可以读写元素的行内样式,但是它无法访问外部 CSS 样式表。如果使用标准的 DOM 方法访问外部样式表,又存在浏览器兼容性的尴尬。jQuery 为设计师解决了这些难题,它把所有样式表操作的难题都包容到几个实用方法中,从而在设计时只需简单调用这几种方法,即可解决 Web 开发中很多界面设计难题。

4.10.1 读写 CSS 样式

CSS 样式存在 3 种形式:行内样式、文档内部样式和文档外部样式。行内样式以元素属性的形式存在,使用 style 属性即可读写。而文档内部样式和文档外部样式统一被视为外部样式,这些外部样式只能够通过 DOM 的 StyleSheets、CSS 和 CSS 2 模块提供的对象、方法和属性才能够访问和操作。

为了方便读写 CSS 样式,0 级 DOM 定义了 Style 对象,并允许 Element 对象通过 style 属性进行访问。Style 对象包含大量 CSS 样式属性,这些属性与 CSS 属性一一对应,但是它们的名称略有不同。详细说明如下:

- ☑ 对于独立单词的 CSS 样式来说,Style 对象以同名来表示对应的 CSS 脚本属性。例如,使用 style.color 可以访问 color 样式。
- ☑ 对于复合词的 CSS 样式来说,由于 CSS 样式使用连字符连接多个单词(如 border-right),但是在 JavaScript 中连字符被约定为减号运算符,如果不进行处理 JavaScript 会误以表达式进行运算。因此,Style 对象的属性名与 CSS 属性名是不同的。如果对应的 CSS 属性名包含一个或多个连字符,则 Style 对象就会删除这些连字符,并以驼峰命名法重命名脚本属性名。例如,对于 border-right 样式属性来说,在脚本中就必须使用 borderRight 脚本属性来访问 border-right 样式属性。
- ☑ 由于 float 是 Java 及其他语言中的关键字,JavaScript 虽然没有把它作为关键字,但作为保留字禁止用户使用。因此,Style 对象没有直接与 float 样式属性对应的脚本属性名。为了解决这个问题,Style 对象在 float 样式属性名前增加了 css 前缀,使用 cssFloat 名来表示与 float 样式属性对应的脚本属性。
- ☑ Style 对象约定所有 CSS 脚本属性值都是字符串,因此在 JavaScript 中为脚本属性赋值时必须加上引号,以字符串数据类型进行传递。
- ☑ 在 CSS 中,样式声明的尾部会添加分号,但是在脚本属性中就不能作为属性值的一部分被引用,因为分号是 JavaScript 的语法组成部分。
- ☑ 当为脚本属性赋值时,必须包含值和完整的单位,省略单位则所设置的脚本样式无效。
- ☑ 在 JavaScript 中可以使用变量动态设置脚本样式的属性值,但是变量的值最后以字符串形式存在,且不要忽略了属性值的单位。

【示例 37】 使用 Style 对象来读取内部样式表属性值以及行内样式属性值，在浏览器中预览则发现只能够读取行内样式。演示效果如图 4.35 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.innerHTML = "color=" + p.style.color + "<br />font-weight=" + p.style.fontWeight
}
</script>
<style type="text/css">
.red { color:red; }
</style>
<title>上机练习</title>
</head>
<body>
<p class="red" style="font-weight:bold">段落文本</p>
</body>
</html>
```

如果使用 JavaScript 直接定义样式，则可以使用 DOM 的 Style 对象直接进行定义。例如：

```
<script type="text/javascript">
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.style.color = "red";           //定义颜色样式
    p.style.fontWeight = "bold";     //定义粗体样式
}
</script>
```

使用 Style 对象可以获取指定元素的行内样式，但是无法获取由 style 元素定义的内部样式表，以及使用 link 元素或 @import 命令导入的外部样式表。要读写 CSS 样式表，可以使用 Document 对象的 styleSheets 对象实现。styleSheets 对象包含了文档中所有样式表的引用。例如，所有 <style> 标签定义的内部样式表，以及使用 link 元素或 @import 命令导入的外部样式表。

DOM 还为每个样式表定义了一个 cssRules 的集合，用来包含指定样式表中所有的规则。但是 IE 浏览器不支持 cssRules 对象，而定义 rules 集合来支持相同的操作。如果需要同时支持 IE 和 Firefox 等主流浏览器，可以使用下面代码进行兼容：

```
var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
```

上面代码先判断浏览器是否支持 cssRules 对象，如果支持则调用 cssRules 对象，否则使用 rules 集合。

cssRules 对象和 rules 集合都包含 style 属性，用来访问 Style 对象，并通过 Style 对象包含的样式脚本属性读写具体的样式。

【示例 38】 文档中仅包含一个样式表，要访问它可以使用 document.styleSheets[0]，而该样式表中仅包含一个样式，所以可以使用 cssRules[0] 进行访问。然后借助 cssRules 对象的 style 属性访问 Style 对象，并读取 color 样式属性的值。演示效果如图 4.36 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```



```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;    //访问样式
    alert(cssRules[0].style.color);                //访问样式属性值，弹出 red 字符串
}
</script>
<style type="text/css">
.red { color:red; }
</style>
<title>上机练习</title>
</head>
<body>
<p class="red" style="font-weight:bold">段落文本</p>
</body>
</html>

```

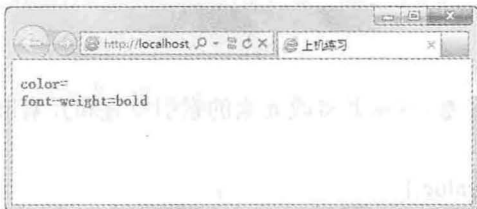


图 4.35 访问行内样式



图 4.36 访问内部样式表

在样式表中写入样式也很简单。例如，以示例 38 为基础，在当前样式中写入一个加粗样式，实现代码如下：

```

<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
    cssRules[0].style.fontWeight = "bold";    //写入样式
}
</script>

```

也可以利用现有样式表创建新的样式，并定义各种属性。例如，在以下示例中为当前样式表添加一个标签样式，并定义背景色为蓝色，字体颜色为白色。由于样式的优先级缘故，文本仍然显示为红色。代码如下：

```

<script type="text/javascript" >
window.onload = function(){
    var n = document.styleSheets[0].length;    //获取当前样式表中包含样式的个数
    if(document.styleSheets[0].insertRule){    //兼容非 IE 浏览器
        document.styleSheets[0].insertRule("p{background-color:blue;color:#fff;}", n); //插入样式
    }else{    //兼容 IE 浏览器
        document.styleSheets[0].addRule("P", "background-color:blue;color:#fff;", n); //添加样式
    }
}
</script>

```

addRule()方法是 IE 浏览器专用方法，它能够为样式表增加一个样式，具体用法如下：

`styleSheet.addRule(selector,style,index)`

`styleSheet` 表示样式表索引对象。

- ☑ 参数 `selector` 表示样式的选择符，必须以字符串的形式传递。
- ☑ 参数 `style` 表示声明样式的字符串，格式与 CSS 样式的格式相同。
- ☑ 参数 `index` 表示索引号，用于设置新建样式在样式表中的索引位置，默认为 -1，表示位于样式表的末尾，该参数可以不设置。

`insertRule()` 方法是非 IE 浏览器专用方法，它也能够为样式表增加一个样式，具体用法如下：

`styleSheet.insertRule(rule ,index)`

- ☑ 参数 `rule` 是一个完整的 CSS 样式字符串，包括选择符和声明两部分，格式必须与 CSS 样式的格式相同。
- ☑ 参数 `index` 与 `addRule()` 方法中的 `index` 参数作用相同，但是该参数必须设置，当值为 0 时表示放置在样式表的末尾。

jQuery 定义了 `css()` 方法，使用该方法能够读取指定的样式，也能够为元素设置 CSS 样式。具体用法如下：

`css(propertyName)`

`css(propertyName, value)`

`css(propertyName, function(index, value))`

`css(map)`

- ☑ 参数 `propertyName` 表示一个 CSS 属性。
- ☑ 参数 `value` 表示一个属性值。
- ☑ 参数 `function(index,value)` 表示一个返回设置值的函数，该函数接收元素的索引位置和元素旧的样式属性值作为参数。
- ☑ 参数 `map` 表示属性名值对构成的对象，如 `{name:value;}`。

【示例 39】 使用 `css()` 方法分别读取 CSS 的 `color` 和 `font-weight` 样式值。演示效果如图 4.37 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").html("color=" + $("p").css("color") + "<br />font-weight=" + $("p").css("font-weight"));
})
</script>
<style type="text/css">
.red {color:red;}
</style>
<title>上机练习</title>
</head>
<body>
<p class="red" style="font-weight:bold">段落文本</p>
</body>
</html>
```

【提示】

通过示例 39 还可以看到，`css()` 方法能够读取指定元素的所有 CSS 样式，不管它是行内样式、内部样式或外部样式。另外，也可以使用 `attr()` 格式设置元素的样式。例如，分别用两种传递参数的方法为 `p` 元素定

义 CSS 样式。注意，当使用对象结构传递样式参数时，样式名称不要加引号，但是样式值应该加引号。使用 `css()` 方法为元素定义样式比较简单。例如，下面分别使用不同的方法为段落文本设置不同的样式，这些样式将以行内样式而存在，代码如下：

```
<script type="text/javascript">
$(function(){
    $("p").css("font-style","italic");           //设置单个样式
    $("p").css({color:"red", fontWeight:"bold"}); //以对象结构的形式传递多个样式
})
</script>
```

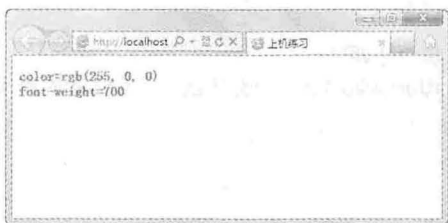


图 4.37 使用 `css()` 方法访问内部样式表

4.10.2 绝对定位

所谓绝对定位就是指定元素距离浏览器窗口左上角的偏移距离。DOM 约定任何元素都拥有 `offsetLeft` 和 `offsetTop` 属性，它们描述了元素的最近偏移位置。但是不同浏览器定义元素的偏移参照对象不同。例如，IE 浏览器总是以父元素为参照对象进行偏移，而非 IE 浏览器会以最近非静态定位元素为参照对象进行偏移。

尽管元素偏移定位存在兼容性问题，但是所有浏览器都支持 `offsetParent` 属性。由于 `offsetParent` 属性总能够自动识别当前元素偏移的参照对象，所以不用担心 `offsetParent` 在不同浏览器中具体指代什么元素。因此，可以不考虑浏览器兼容性问题，通过迭代计算当前元素距离窗口左上角，而不去关心各个浏览器需要经过几次偏移定位才能够到达窗口左上角。

当然，jQuery 帮助用户简化了这个迭代操作。jQuery 定义了 `offset()` 方法，该方法能够获取匹配元素在当前视口的相对偏移。具体用法如下：

```
offset()
offset(coordinates)
offset(function(index, coords))
```

- ☑ 参数 `coordinates` 表示一个对象，包含 `top` 和 `left` 属性，用整数指明元素的新顶部和左边坐标。
- ☑ 参数函数 `function(index, coords)` 返回用于设置坐标的一个函数，该参数函数接收元素在匹配的元素集合中的索引位置作为第 1 个参数和当前坐标作为第 2 个参数，这个函数应该返回一个包含 `top` 和 `left` 属性的对象。

如果调用 `offset()` 方法没有传递参数，则将返回为一个对象，包含 `top` 和 `left` 两个属性，用于分别存储匹配元素的顶部偏移和左侧偏移。注意，该方法仅对可见元素有效。

【示例 40】 演示如何获取 3 个 `div` 元素的绝对偏移位置，演示效果如图 4.38 所示。为了方便比较和观察，在样式表中定义页边距为 0，并定义 `div` 元素的大小固定，且边框都为 10 像素宽。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
```



```

<script type="text/javascript" >
$(function(){
    var o1 = $("div").eq(0).offset();           //获取第 1 个 div 元素的偏移信息
    $("div").eq(0).html( "left: " + o1.left + "<br />top: " + o1.top ); //显示信息
    var o2 = $("div").eq(1).offset();           //获取第 2 个 div 元素的偏移信息
    $("div").eq(1).html( "left: " + o2.left + "<br />top: " + o2.top ); //显示信息
    var o3 = $("div").eq(2).offset();           //获取第 3 个 div 元素的偏移信息
    $("div").eq(2).html( "left: " + o3.left + "<br />top: " + o3.top ); //显示信息
})
</script>
<style type="text/css">
body { padding:0; margin:0; }/*清除页边距*/
div {height:60px; width:200px; border:solid 10px red; }/*统一 div 元素的显示样式*/
</style>
<title>上机练习</title>
</head>
<body>
<div>盒子 1</div>
<div style="float:left">盒子 2</div>
<div style="float:left">盒子 3</div>
</body>
</html>

```

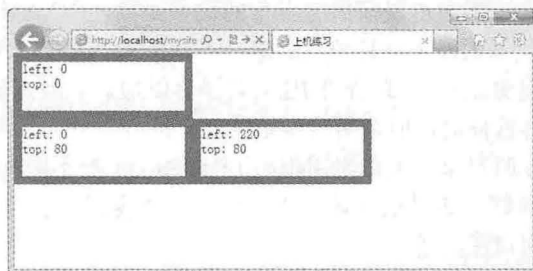


图 4.38 获取元素的绝对偏移位置

注意，offset()方法允许用户重新设置元素的位置，这个元素的位置是相对于 document 对象的。如果对象原先的 position 样式属性是 static，会被改成 relative 来实现重定位。

4.10.3 相对定位

所谓相对位置就是获取指定元素距离最近父级定位元素左上角的偏移距离。这里读者首先需要弄明白以下两个概念：

- ☑ 定位元素就是被定义了相对、绝对或固定定位的元素，即设置了 CSS 的 position 属性值为 absolute、fixed 或 relative 属性值的元素。
- ☑ 所谓父元素是与当前元素相邻的上一级元素，而最近的父级元素就不一定与当前元素相邻，也可能距离很远。如果当前元素的上级元素都没有被定义 position 属性值为 absolute、fixed 或 relative，则当前元素的最近父级定位元素就应该是 body 元素了，此时相对偏移位置与绝对偏移位置是相同的。

使用 JavaScript 实现获取指定元素的相对偏移位置稍显麻烦，但是其设计思路比较简单，设计的核心就是利用 offsetParent 属性获取最近的父级定位元素，然后判断该元素的位置。如果它是父元素，则可以直接读取当前元素的 offsetLeft 和 offsetTop 属性值。如果不是父元素，则可以获取当前元素的绝对偏移位置减去定位元素的绝对偏移位置，即可获得当前元素距离定位元素的偏移距离。

同时,由于不同浏览器对于 `offsetLeft` 和 `offsetTop` 属性值的解析方式不同,当直接调用该属性读取相对偏移位置时,可能存在误差。其中,IE 浏览器会加上父元素的边框,因此还必须使用 `offsetLeft` 和 `offsetTop` 属性值减去父元素的边框值。如果当元素定义了 `border` 样式后,可以直接读取该样式值。但是如果元素没有显示定义父元素的边框,或者通过脚本动态设置了元素的边框,就无法读取这个值,此时就必须根据不同浏览器提供的私有解决方法分别进行读取。

jQuery 定义了 `position()` 方法,使用该方法可以获取匹配元素的相对偏移位置。具体用法如下:

`position()`

`position()` 方法的用法与 `offset()` 方法相同,都返回一个包含两个属性(即 `top` 和 `left`)的对象。注意,为精确计算结果,请在补白、边框和填充属性上使用像素单位,该方法只对可见元素有效。

获取匹配元素中第 1 个元素的当前坐标,相对于 `offset parent` 的坐标。`offset parent` 指离该元素最近的而且被定位过的祖先元素。`position()` 方法可以获得该元素相对于 `offset parent` 的当前坐标。与 `offset()` 不同,`offset()` 是获得该元素相对于 `document` 的当前坐标,当把一个新元素放在同一个容器里面另一个元素附近时,`position()` 更好用。

【示例 41】 分别定义两个 `div` 元素,一个直接放在文档中,另一个包裹在被定义了相对定位的盒子中,同时设置这个盒子向右浮动。最后使用 `position()` 方法读取这两个 `div` 元素的相对偏移位置。演示效果如图 4.39 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var o1 = $("div").eq(0).position();           //获取元素的相对偏移位置
    $("div").eq(0).html( "left: " + o1.left + "<br />top: " + o1.top );      //显示相对偏移位置
    var o2 = $("div").eq(2).position();           //获取元素的相对偏移位置
    $("div").eq(2).html( "left: " + o2.left + "<br />top: " + o2.top );      //显示相对偏移位置
})
</script>
<style type="text/css">
body { padding:0; margin:0; }
div { height:60px; width:200px; border:solid 10px red; }
</style>
<title>上机练习</title>
</head>
<body>
<div>盒子 1</div>
<div style="position:relative; float:right; width:300px; height:100px; border-color:blue;">
    <div>盒子 2</div>
</div>
</body>
</html>
```

在示例 41 中如果取消定义盒子元素的行内样式 `position:relative`,则可以看到两个 `div` 元素都会以窗口左上角为坐标原点进行定位测量。演示效果如图 4.40 所示。

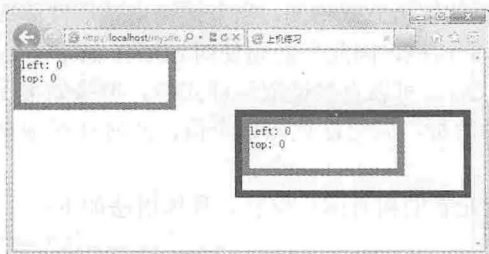


图 4.39 为盒子元素定义相对定位后的效果

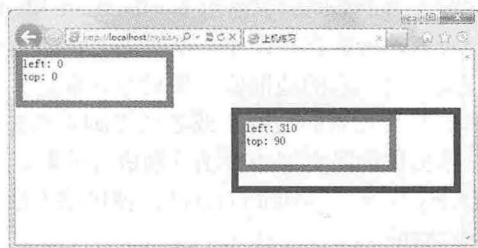


图 4.40 没有为盒子元素定义相对定位后的效果

4.10.4 设置大小

在 Web 开发中,经常需要控制元素的大小。使用 CSS 技术可以直接设置大小,但是如果没有显式定义元素的宽和高,获取元素大小就比较麻烦。jQuery 定义了 `width()` 和 `height()` 方法,利用这两个方法可以很轻松地读写元素的大小。具体用法如下:

```
width()
width(value)
width(function(index,width))
height()
height(value)
height(function(index,width))
```

- ☑ 参数 `value` 表示一个正整数代表的像素数,或整数和一个可选的附加单位(默认是 `px`) (作为一个字符串)。
- ☑ 参数函数 `function(index,width)` 返回用于设置的宽度,该函数接收元素的索引位置和元素旧的高度值作为参数。

【示例 42】 演示如何读取元素的大小,以及如何动态设置元素的大小。演示效果如图 4.41 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").html("height (原) = " + $("div").height() + "<br />width (原) = " + $("div").width());
    //获取元素设置前的宽和高
    $("div").height(140); //重设元素高度为 140px
    $("div").width("30em"); //重设元素宽度为 30em
    $("div").html($("div").html() + "<br />height (现) = " + $("div").height() + "<br />width (现) = " + $("div").width());
    //获取元素设置前的宽和高
})
</script>
<title>上机练习</title>
</head>
<body>
<div style="border:solid 10px red;">盒子</div>
</body>
</html>
```


width()和 height()方法在没有传递参数时,表示读取元素的宽度和高度,返回值的单位为像素。也可以传递参数来设置元素的宽和高,如果直接传递一个数值,则默认单位为 px。也可以以字符串形式传递值和单位。

除了 height()和 width()方法, jQuery 还定义了 innerHeight()、innerWidth()、outerHeight()和 outerWidth()方法。这些方法实际上是在 height()和 width()方法基础上,计算元素的边框或补白。其中, outerHeight()和 outerWidth()方法能够返回元素的总宽和总高(包括宽高、补白和边框宽度), innerHeight()和 innerWidth()方法能够返回元素的内容宽度和高度(包括宽高和补白)。

【示例 43】 分别演示如何计算元素的总宽、总高、内容宽度和内容高度。演示效果如图 4.42 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").html("innerHeight=" + $("div").innerHeight() + "<br />innerWidth=" + $("div").innerWidth());
    $("div").html( $("div").html() + "<br />outerHeight=" + $("div").outerHeight() + "<br />outerWidth=" +
    $("div").outerWidth());
})
</script>
</script>
<style type="text/css">
div { width:200px; height:50px; margin:50px; padding:50px; border:solid 50px red; }
</style>
<title>上机练习</title>
</head>
<body>
<div style="border:solid 10px red;">盒子</div>
</body>
</html>
```

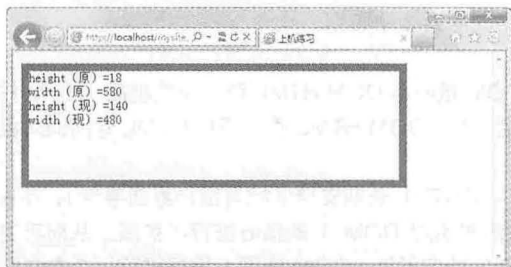


图 4.41 读写元素的宽和高

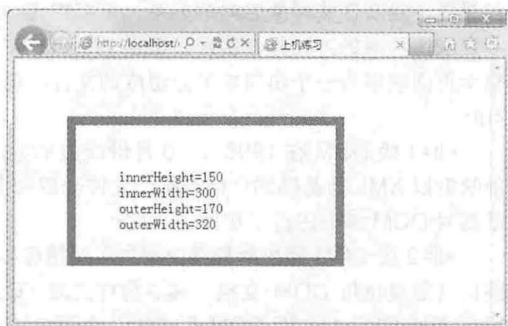


图 4.42 获取元素的总宽、总高、内容宽度和内容高度

4.11 访问文档树

DOM 为 Node 类型定义了 childNodes、parentNode、nextSibling、previousSibling、firstChild 和 lastChild 属性,这些属性为节点之间相互访问提供了支持。开发人员正是利用这些指针属性,可以自由地在文档结

构内进行访问。

由于 DOM 提供的这些指针属性是针对节点而言的,但是节点类型又有很多种,而在实际开发中开发人员更多的是需要遍历元素,而不是遍历文本或注释等类型节点。因此,直接使用这些属性有时会感到不方便。jQuery 考虑到开发人员的实际需求,对这些属性进行了包装,以方便用户快速访问。

jQuery 定义了 children()、next()、prev()、parent() 4 种基本元素遍历方法,使用它们可以轻松访问文档中的任何元素。其中,children()方法用于获取当前元素包含的所有子元素,next()方法用于获取当前元素相邻的下一个同级元素,prev()方法用于获取当前元素相邻的上一个同级元素,parent()方法用于获取当前元素的父元素,不过这些方法返回值都是 jQuery 对象,而不是 DOM 集合或对象。

【示例 44】 借助 jQuery 定义的基本指针函数,从 body 元素开始,沿着 DOM 结构树,一步步访问到 li 元素,并修改文档中 3 个 li 元素包含的文本内容。演示效果如图 4.43 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var $body = $("body");           //获取 body 元素
    var li = $body.children().eq(2).children()[0]; //利用 children()方法,遍历到第一个 li 元素
    $(li).text("第 1 项").next().text("第 2 项").next().text("第 3 项"); //利用 next()方法,遍历 li 元素,并修改每个 li 元素的文本内容
})
</script>
<title>上机练习</title>
</head>
<body>
<h1>DOM</h1>
<p>DOM 实际上是以面向对象方式描述的文档模型。DOM 定义了表示和修改文档所需的对象、这些对象的行为和属性,以及这些对象之间的关系。可以把 DOM 理解为是页面上数据和结构的一个树形表示,不过页面当然并不是以这种树的方式具体实现。根据 W3C DOM 规范,DOM 是 HTML 与 XML 的应用编程接口(API),DOM 将整个页面映射为一个由层次节点组成的文件,包括 1 级、2 级、3 级共 3 个级别。</p>
<ul>
<li>1 级 DOM 在 1998 年 10 月份成为 W3C 的提议,由 DOM 核心与 DOM HTML 两个模块组成。DOM 核心能映射以 XML 为基础的文档结构,允许获取和操作文档的任意部分。DOM HTML 通过添加 HTML 专用的对象与函数对 DOM 核心进行了扩展。</li>
<li>2 级 DOM 通过对象接口增加了对鼠标和用户界面事件(DHTML 长期支持鼠标与用户界面事件)、范围、遍历(重复执行 DOM 文档)和层叠样式表(CSS)的支持。同时也对 DOM 1 的核心进行了扩展,从而可支持 XML 命名空间。2 级 DOM 引进了几个新 DOM 模块来处理新的接口类型:DOM 视图,描述跟踪一个文档的各种视图(使用 CSS 样式设计文档前后)的接口;DOM 事件,描述事件接口;DOM 样式,描述处理基于 CSS 样式的接口;DOM 遍历与范围,描述遍历和操作文档树的接口。</li>
<li>3 级 DOM 通过引入统一方式载入和保存文档和文档验证方法对 DOM 进行进一步扩展,DOM 3 包含一个名为“DOM 载入与保存”的新模块,DOM 核心扩展后可支持 XML 1.0 的所有内容,包括 XML Infoset、XPath 和 XML Base。</li>
</ul>
</body>
</html>
```

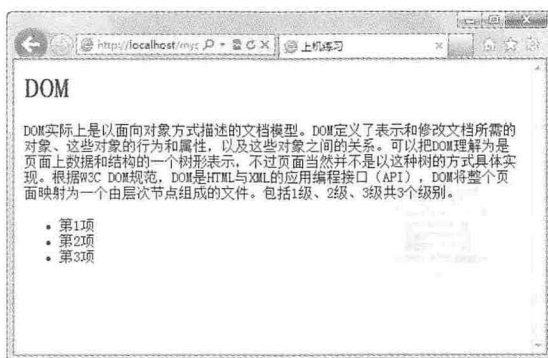


图 4.43 使用 jQuery 遍历指针方法

`children()`方法返回的是一个集合，因此可以使用 jQuery 的方法进行筛选，也可以使用数组方法直接访问其中的元素。如果使用数组方法访问时，就需要使用 `$()` 方法把访问得到的元素对象转换为 jQuery 对象，否则无法调用 jQuery 的方法。

第 5 章

事件处理

( 视频讲解：1 小时 30 分钟)

用户与网页交互时产生的操作称为事件。事件可以由用户引发，也可能是页面发生改变，甚至还有用户看不见的事件，如 Ajax 的交互进度改变。绝大部分事件都由用户的动作所引发，如按下鼠标按键时，产生 click 事件；移动鼠标指针，产生 mouseover 事件等。在 JavaScript 中，事件往往与事件处理函数配合使用。由于早期浏览器都有自己的事件处理模式，导致 Web 开发中事件处理标准不统一，后来 W3C 组织规范了事件处理模式，使事件处理逐步走上标准。

经过了漫长的演变史，开发人员已经告别了内嵌式的事件处理方式（即直接将事件处理函数放在 HTML 元素内使用）。如今事件已是 DOM 的重要组成部分，但遗憾的是，IE 继续保留它最早在 IE 4.0 中实现的事件模型，以后的 IE 版本中也没有做太大的改变。也就是说，IE 使用的还是一种专有的事件模型，而其他的主流浏览器直到 DOM 级别 3 规定定案后，才陆陆续续支持 DOM 标准的事件处理模型。不过，jQuery 通过技术封装帮助我们解决了这个问题，使事件开发变得轻松和便捷许多。

5.1 事件处理模型

事件模型也就是事件驱动模式或方式，它是面向对象程序设计的核心之一，以消息为基础，以事件来驱动。当文档发生了特定事件时，浏览器会自动生成一个事件对象（Event），以响应该事件，并执行对应的 JavaScript 脚本。

在各大主流浏览器中存在 3 种事件模型：原始事件模型、DOM 事件模型和 IE 事件模型。其中原始事件模型被所有浏览器支持，而 DOM 中所定义的事件模型目前被除了 IE 以外的所有主流浏览器支持。

5.1.1 原始事件模型

原始事件模型是由 Navigator 浏览器最早引入的，因此人们习惯上称之为 0 级事件模型。所有现代浏览器都支持这种模型，它是应用最广泛、影响最深远的事件处理模型。

在 0 级事件模型中，事件处理程序被定义为函数实例，然后绑定到 DOM 元素的事件属性上面，从而实现事件注册。例如，绑定函数到 onclick 属性上，用来处理 click（鼠标单击）事件；绑定函数到 onmouseover 属性上，用来处理 mouseover（鼠标移过）事件等。

【示例 1】 定义一个匿名函数，然后把它赋值给按钮的 onclick 属性，实现注册事件。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    btn.onclick = function()           //绑定事件处理函数，为按钮注册鼠标单击事件
        alert(this.nodeName);
    }
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="0 级事件模型" />
</body>
</html>

```

0 级事件模型允许把特定的事件处理函数的函数体直接赋值给 DOM 元素的 HTML 特性，从而简化了事件注册的程序。例如，针对示例 1 可以简写为下面一行代码。

```
<input type="button" onclick="alert(this.nodeName);" value="0 级事件模型" />
```

0 级事件模型使用起来虽然很方便，但是存在致命的缺陷：元素属性被用来存储事件处理函数的引用，所以每个元素对于任何特定事件类型，每次只能注册一个事件处理程序。如果独立管理事件流会显得非常笨拙，甚至无能为力。

5.1.2 DOM 事件模型

W3C 规范在 DOM 级别 1 中并没有定义任何的事件，直到发布于 2000 年 11 月的 DOM 级别 2 才定义了一小分子集，DOM 级别 2 中已经提供了一种更详细、更细致的方式以控制 Web 页面中的事件。最后，完整的事件是在 2004 年 DOM 级别 3 的规定中才最终定案。因为 IE 4.0 是 1995 推出的并已实现了自己的事件模型（冒泡型），当时根本就没有 DOM 标准，不过在以后的 DOM 标准规范过程中已经把 IE 的事件模型吸收到了其中。

目前，除 IE 浏览器外，其他主流的 Firefox、Opera、Safari 都支持标准的 DOM 事件处理模型。IE 仍然使用自己专有的事件模型，即冒泡型，它的事件模型的一部分被 DOM 标准采用，这点对于开发者来说也是有好处的，只有使用 DOM 标准，IE 共有的事件处理方式才能有效地跨浏览器。

1. 注册事件

在 DOM 2 级事件模型中为 DOM Element 类型元素都定义了 `addEventListener()` 方法，通过这个方法注册事件，放弃使用 0 级事件模型中为元素的事件属性指定事件处理函数的方法。`addEventListener()` 方法的具体用法如下：

```
addEventListener(type, function, useCapture)
```

- ☑ 参数 `type` 表示要绑定的事件类型，事件类型与事件属性不同，它没有前缀 `on`。例如，对于事件属性 `onclick` 来说，对应事件类型为 `click`。
- ☑ 参数 `function` 表示调用的事件处理函数，该函数自带一个默认参数，引用 `Event` 对象，以方便传递事件发生时的相关操作信息。
- ☑ 参数 `useCapture` 为一个布尔值。如果为 `true`，则在事件传播的捕捉阶段触发响应；如果该参数值为

false, 则在事件传播的冒泡阶段触发响应。

【示例 2】 为按钮注册一个鼠标单击事件类型, 然后在事件处理函数中获取当前事件的 Event 对象, 并显示当前事件的类型。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0]; //获取按钮元素
    btn.addEventListener("click", function(event){ //注册鼠标单击事件, 并设置事件流为捕获型事件
        var event = event || window.event; //兼容 Event 对象
        btn.value = event.type; //显示当前事件的类型
    },true);
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象" />
</body>
</html>
```

2. 事件传播

在 DOM 2 级事件模型中, 一旦事件被触发, 事件流首先从 DOM 树顶部 (文档节点) 向下传播, 直到目标节点, 然后再从目标节点向上传播到 DOM 树顶。从上到下的过程被称为捕获阶段, 从下到上的过程被称为冒泡阶段。

addEventListener() 方法的第 3 个参数可以设置事件响应的阶段。如果参数值为 true, 则事件只在捕获阶段被触发, 可以标志为捕获型处理程序; 如果参数值为 false, 则事件只在冒泡阶段被触发, 可以标志为冒泡型处理程序。

【示例 3】 利用循环体结构分别为按钮元素及其所有父级节点注册一个捕获型鼠标单击类事件处理函数。在浏览器中预览, 当单击按钮时, 可以看到事件触发的先后过程: 首先触发的是 #document 节点, 然后是 HTML 节点、BODY 节点, 最后才是 INPUT, 如图 5.1 所示。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0]; //获取按钮
    var p = document.getElementsByTagName("p")[0]; //p 元素
    var i = 1; //声明并初始化一个临时变量
    do{ //使用 do 循环结构逐层注册鼠标单击事件
        btn.addEventListener("click", function(){ //注册鼠标单击事件
            p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
        },true); //动态跟踪当前响应节点的名称
        btn = btn.parentNode; //访问上一级父元素
    } while(btn); //设置循环条件, 如果存在父节点
```



```

}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象">
<p>捕获型事件流传播过程: </p>
</body>
</html>

```

下面修改 `addEventListener()` 方法的第 3 个参数, 设置参数值为 `false`, 即注册事件为冒泡型处理程序。然后在浏览器中预览, 当单击按钮时, 可以看到事件触发的先后过程: 首先触发的是 `INPUT` 节点, 然后是 `BODY` 节点、`HTML` 节点, 最后才是 `#document` 节点, 如图 5.2 所示。

```

<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0]; //获取按钮
    var p = document.getElementsByTagName("p")[0];       //p 元素
    var i = 1;                                           //声明并初始化一个临时变量
    do{                                                  //使用 do 循环结构逐层注册鼠标单击事件
        btn.addEventListener("click", function(){      //注册鼠标单击事件
            p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
        },false);                                     //动态跟踪当前响应节点的名称
        btn = btn.parentNode;                          //访问上一级父元素
    } while(btn);                                       //设置循环条件, 如果存在父节点
}
</script>

```



图 5.1 捕获型事件流传播过程

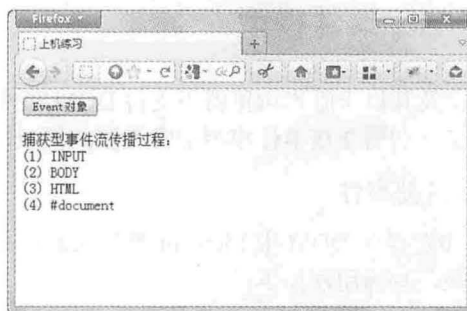


图 5.2 冒泡型事件流传播过程

3. 销毁事件

2 级事件模型还为 DOM 中 `Element` 类型元素定义了 `removeEventListener()` 方法, 通过这个方法可以随时销毁已经注册的事件, 以节省系统资源。该方法的用法与 `addEventListener()` 方法相同。

【示例 4】针对示例 3, 可以在事件处理函数中为当前对象添加 `removeEventListener()` 方法, 销毁刚刚注册的鼠标单击事件, 避免事件处理函数被多次触发。其中, `arguments.callee` 引用当前匿名的事件处理函数。

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0]; //获取按钮

```

```

var p = document.getElementsByTagName("p")[0];           //p 元素
var i = 1;                                               //声明并初始化一个临时变量
do{                                                      //使用 do 循环结构逐层注册鼠标单击事件
    btn.addEventListener("click", function(){           //注册鼠标单击事件
        p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
    },false);                                           //动态跟踪当前响应节点的名称
    this.removeEventListener("click",arguments.callee,false); //注销当前鼠标单击事件
    btn = btn.parentNode;                               //访问上一级父元素
} while(btn);                                           //设置循环条件，如果存在父节点
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象" />
<p>冒泡型事件流传播过程：</p>
</body>
</html>

```

【提示】

removeEventListener()方法的第3个参数值与addEventListener()方法的第3个参数值保持一致。也就是说，如果使用addEventListener()方法将事件处理函数绑定到捕获阶段，则必须在removeEventListener()方法中指明捕获阶段，只有这样才能够正确地删除事件处理函数。如果绑定在冒泡阶段的事件处理函数，而在捕获阶段来删除它，虽然这样做不会引发错误，但是销毁操作是无效的。

5.1.3 IE 事件模型

IE 7 及其以下版本浏览器不支持 DOM 事件模型，在 IE 8 版本中才开始支持。不过 IE 从早期版本开始就提供了一套与 2 级事件模型非常相似的模型，虽然用法略有区别，但功能基本相同。

1. 注册事件

IE 浏览器为 DOM 中 Element 类型元素定义了 attachEvent()方法，通过这个方法注册事件，该方法包含两个参数，具体用法如下：

attachEvent(type, function)

☑ 参数 type 用来设置元素的事件属性，如 onclick，而不是 2 级事件模型中的 click 事件类型。

☑ 参数 function 与 addEventListener()方法中的第 2 个参数相同，都是指事件处理函数。

【示例 5】如果完全兼容 IE 与非 IE 浏览器，则在注册事件时可以借助 if 条件语句分别进行注册。针对示例 2，可以按如下方法进行修改，实现在不同浏览器下都可以正常注册。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    var p = document.getElementsByTagName("p")[0];
    var i = 1;
    do{

```

```

if(btn.addEventListener) //如果支持 addEventListener()方法, 则调用该方法
    btn.addEventListener("click", function(){
        p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
    },false);
else{ //否则, 调用 attachEvent()方法在 IE 下注册事件
    btn.attachEvent("onclick", (function(btn){
        return function(){ //返回闭包函数, 从而动态锁定响应事件的当前对象
            p.innerHTML += "<br />(" + i++ + ") " + btn.nodeName;
        }
    })(btn)); //直接调用函数, 以便向内部传递当前绑定元素对象
}
btn = btn.parentNode;
} while(btn);
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象" />
<p>事件流传播过程: </p>
</body>
</html>

```

【提示】

当使用 `attachEvent()` 方法注册事件时, 在事件处理函数中的 `this` 指针总是指向 `Window` 对象。但是在 0 级事件模型中, 事件处理函数中的 `this` 指针总是指向当前注册事件的对象。而在 DOM 2 级事件模型中, `addEventListener()` 方法中的 `this` 都会指向触发当前事件的对象。为了解决 `attachEvent()` 方法无法定位当前事件对象的问题, 可以通过返回闭包函数的形式, 在 `attachEvent()` 方法中的第 2 个参数中直接调用函数, 而不是引用函数, 然后通过返回闭包函数以实现传递事件处理函数。这样在调用函数时, 把当前事件对象传递给闭包函数, 从而动态锁定事件对象。示例 5 分别在 Firefox 和 IE 下预览, 效果如图 5.3 和图 5.4 所示, 执行效果完全相同。

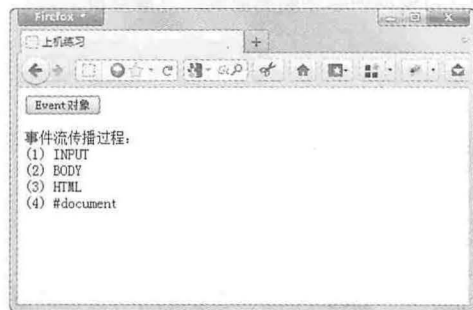


图 5.3 在 Firefox 浏览器下的预览效果

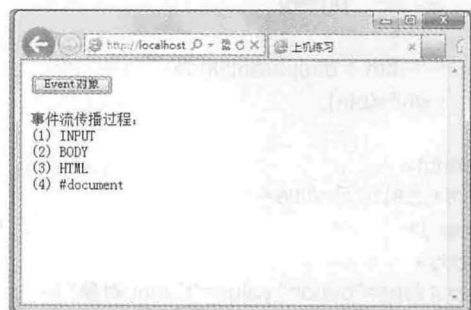


图 5.4 在 IE 浏览器下的预览效果

2. 事件传播

在 IE 浏览器中, 事件流总是从目标对象向上逐层传递到 DOM 树顶。在 2 级事件模型中, 事件冒泡只适用于原始事件和输入类事件, 如鼠标事件、键盘事件, 不支持高级语义事件。IE 的事件冒泡传播与 2 级事件模型是相同的, 但是它们中止冒泡的方式不同。

2 级事件模型通过 `stopPropagation()` 方法可以中止事件流的冒泡, 而 IE 事件模型设计通过 `Event` 对象的 `cancelBubble` 属性来控制。如果要控制事件流继续冒泡, 则可以定义:


```
window.event.cancelBubble = true;
```

注意, cancelBubble 属性只适用当前事件。当新事件发生时, 将被赋予给新的 Event 对象, 此时 cancelBubble 属性值被还原为默认值 false。

3. 注销事件

IE 支持使用 detachEvent()方法注销事件, 该方法也包含有两个参数: 第 1 个参数为事件属性名(如 onclick), 第 2 个参数为事件处理函数。

【示例 6】设计当按钮被单击一次之后, 使用 detachEvent()方法中止当前对象绑定的鼠标单击事件。这样按钮就不只能够被单击一次, 单击一次之后自动失效。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    var p = document.getElementsByTagName("p")[0];
    var i = 1;
    do{
        if(btn.addEventListener)                //如果支持 addEventListener()方法, 则调用该方法
            btn.addEventListener("click", function(){
                p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
                this.removeEventListener("click",arguments.callee,false);
            },false);
        else{                                    //否则, 调用 attachEvent()方法在 IE 下注册事件
            btn.attachEvent("onclick", (function(btn){
                return function(){                //返回闭包函数, 从而动态锁定响应事件的当前对象
                    p.innerHTML += "<br />(" + i++ + ") " + btn.nodeName;
                    btn.detachEvent("onclick", arguments.callee);    //注销 IE 事件
                }
            })(btn));                                //直接调用函数, 以便向内部传递当前绑定元素对象
        }
        btn = btn.parentNode;
    } while(btn);
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象" />
<p>事件流传播过程: </p>
</body>
</html>
```

5.2 事件处理机制

Event 是 JavaScript 中的重要事件, event 代表事件的状态, 专门负责对事件的处理, 它的属性和方法能帮助用户完成各种交互操作。当了解了主流事件模型之后, 下面就来深入认识事件处理的机制。

5.2.1 Event 对象

当触发事件时,浏览器会自动创建一个 Event 对象,Event 对象实际上是 Event 类型实例,默认状态下它会被作为参数传递给事件处理函数。IE 浏览器把 Event 视为独立的对象,通过 Window 对象的 event 属性进行访问。为了能够兼容 IE 与非 IE 浏览器共享 Event 对象,可以使用如下代码进行兼容:

```
var event=event||window.event;           //兼容不同类型的浏览器访问 Event 对象方法
```

Event 对象包含了属性和方法,利用这些属性可以动态存储与当前事件相关的信息,如响应事件的类型、按下的鼠标键、按下的键盘键、光标指针的位置等。

【示例 7】 演示 Event 对象在事件处理中扮演的角色。如果单击按钮时,则会在按钮上提示当前单击的事件类型;如果双击,则会显示双击事件类型。演示效果如图 5.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    btn.onclick = function(event){           //绑定鼠标单击事件处理函数
        var event = event || window.event;   //兼容不同类型浏览器
        btn.value = event.type;               //获取当前事件的类型
    }
    btn.ondoubleclick = function(event){      //绑定鼠标双击事件处理函数
        var event = event || window.event;   //兼容不同类型浏览器
        btn.value = event.type;               //获取当前事件的类型
    }
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="Event 对象" />
</body>
</html>
```



图 5.5 Event 对象在事件中的作用

【提示】

IE 和 DOM 标准浏览器虽然都使用同名的 Event 来存储事件的动态信息,但是它们的属性和方法存在差异,即使是同名的属性,返回的属性值也可能解释不同。例如,IE 使用 srcElement 属性记录当前事件作用的元素,而 DOM 标准使用 target 属性记录当前事件作用的元素。IE 和 DOM 标准都支持 button 属性,但是

IE 的 button 属性返回值为 1、2、4，而 DOM 标准的 button 属性返回值为 0、2、1，这些值分别表示鼠标的左、右和中键。IE 与标准事件模型的 Event 对象详细比较说明如表 5.1 所示。

表 5.1 IE 与标准事件模型的 Event 对象详细比较

IE 事件属性	说 明	标准事件属性	说 明
altKey	true 表示按下了 Alt 键, false 表示没有	altKey	true 表示按下了 Alt 键, false 表示没有
ctrlKey	true 表示按下了 Ctrl 键, false 表示没有	ctrlKey	true 表示按下了 Ctrl 键, false 表示没有
shiftKey	true 表示按下了 Shift 键, false 表示没有	shiftKey	true 表示按下了 Shift 键, false 表示没有
button	鼠标事件。0 为没有按下鼠标键, 1 为按下左键, 2 为按下右键, 4 为中间键, 3 为同时按下左右键, 5 为按下左键和中键, 6 为按下右键和中键, 7 为按下左键、中键、右键	button	0 为左键, 1 为中键, 2 为右键
clientX	事件发生时, 鼠标在浏览器窗口 (不包含工具栏、滚动条等) 的 x 坐标	clientX	事件发生时, 鼠标在浏览器窗口 (不包含工具栏、滚动条等) 的 x 坐标
clientY	事件发生时, 鼠标在浏览器窗口 (不包含工具栏、滚动条等) 的 y 坐标	clientY	事件发生时, 鼠标在浏览器窗口 (不包含工具栏、滚动条等) 的 y 坐标
screenX	事件发生时, 鼠标在整个屏幕上的 x 坐标	screenX	事件发生时, 鼠标在整个屏幕上的 x 坐标
screenY	事件发生时, 鼠标在整个屏幕上的 y 坐标	screenY	事件发生时, 鼠标在整个屏幕上的 y 坐标
type	事件的名称 (如 click)	type	事件的名称 (如 click)
srcElement	引起事件的元素	target	引起事件的元素
keyCode	对于 keypress 事件, 表示按钮的 unicode 字符; 对于 keydown 和 keyup 事件, 则表示按钮的数字代码	charCode	表示按键的 Unicode 字符
		keyCode	表示按键的数字代码
cancelBubble	值为 true 时将阻止事件继续向上冒泡	stopPropagation	可以调用这个方法阻止事件继续向上冒泡
		cancelBubble	true 表示事件冒泡已被取消, false 表示没有
returnValue	值为 false 时将会阻止事件的默认行为	preventDefault()	调用该方法可以阻止事件的默认行为
offsetX	获取事件发生时鼠标相对于引起事件的元素的 x 坐标, 即以引起事件的元素的本身 (不用计算 padding 和 margin) 的左上角为原点	layerX	当引发事件的元素没有动态定位时, 返回鼠标相对于引发事件的元素的最近一个设置了动态定位的父元素里的 x 坐标, 以其父元素的边框的左上角为原点。 当引发事件的元素设置了动态定位后, 则返回鼠标相对于引发事件的元素的 x 坐标, 以该元素边界的左上角为原点
x	获取鼠标相对于引发事件的元素的最近一个设置了动态定位的父元素的 X 坐标, 以该父元素的边框 i 的左上角为原点		

5.2.2 事件流

事件处理模型可以分为基本事件处理和高级事件处理两种。原始事件模型属于基本事件处理, 标准事件模型和 IE 事件模型属于高级事件处理。

基本事件处理主要是指原始事件模型实现的事件处理, 其主要分为以下两种形式:

☑ 作为 HTML 标签属性的事件处理

例如, `<div onmouseover="var a=1; alert();"></div>` 中的 `onmouseover`, 赋给 `onmouseover` 等事件处理函数的是 JS 代码串, 系统会把这些代码串自动包装在一个匿名函数中, 其中可以有 `this` 关键字, 它指向的是这个标签元素, 也可以有 `event` 关键字, 它表示的是事件发生时的事件对象。

☑ 作为 JavaScript 属性的事件处理

例如 `element.onmouseover=function(){……}`。注意, 在这种方式中不能再给事件处理函数赋值 JS 代码串, 而是直接把函数 (不是函数调用) 赋给它, 或是赋一个匿名函数, 例如 `element.onmouseover=function(){};` 或者 `element.onmouseover=f`, 其中 `f` 为一个函数, 在这里不加括号, 基本事件处理也会以冒泡的形式向上传播。

高级事件处理主要是指标准事件模型和 IE 事件模型实现的事件处理。高级事件引入了事件流, 把事件传播分为以下两种类型:

☑ 捕获传播, 即事件从外传到里, 每一级都发生了该事件。

☑ 冒泡传播, 即事件从里传到外, 每一级都发生了该事件, 并不是所有的事件都会冒泡。

在 DOM 文档结构树中, 每个元素并非都孤立的存在, 上下级元素之间必然存在嵌套关系, 这就给 Web 的事件处理带来很多麻烦。当触发 DOM 树上某个元素的事件时, 浏览器的事件处理机制会检查在那个元素上是否已经建立了特定的事件处理程序。如果是, 则调用该事件处理程序, 但是事情远远没有结束。

在目标元素获取机会处理事件之后, 事件模型会检查目标元素的父元素, 检测父元素是否也注册了同类型的事件。如果是, 还要调用父元素的事件处理程序。在这之后, 事件模型还会继续检测上一级元素, 以此逐层检索, 持续不停直到 DOM 树的顶部。这个事件检测的过程如同水中向上传播的气泡, 故有人形象地把这个事件处理过程称为事件冒泡, 如图 5.6 所示。

【示例 8】 在本示例中分别为层层嵌套的多个 `div` 元素注册 `mouseover` 和 `mouseout` 事件。在 `mouseover` 的事件处理程序中动态设置当前 `div` 元素的边框颜色为红色, 在 `mouseout` 的事件处理程序中动态设置当前 `div` 元素的边框颜色为白色。这样当鼠标在页面中移动时, 可以看到 `div` 元素的边框呈波浪形变色或者显隐。演示效果如图 5.7 所示。



图 5.6 事件冒泡示意图

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
```

```
<script type="text/javascript">
```

```
window.onload = function(){
```

```
    var div = document.getElementsByTagName('div'); //获取所有 div 元素
```

```
    for (var i = 0; i < div.length; ++i){ //遍历 div 元素
```

```
        div[i].onmouseover = (function(i){ //依次为每个 div 元素注册鼠标经过事件
```

```
            return function(){ //以闭包形式存储动态变量 i 的值, 以便定位 div
```

```
                div[i].style.borderColor = 'red'; //定义边框的颜色样式为红色
```

```
            }
```

```
        })(i);
```

```
        //向闭包内传递变量 i 的值
```

```
    }
```

```
    for (var i = 0; i < div.length; ++i){ //遍历 div 元素
```

```
        div[i].onmouseout = (function(i){ //依次为每个 div 元素注册鼠标移出事件
```

```
            return function(){ //以闭包形式存储动态变量 i 的值, 以便定位 div
```

```
                div[i].style.borderColor = 'white'; //定义边框的颜色样式为白色
```

```

    }
    })(i);
}
</script>
<style type="text/css">
div {margin:12px 10px; border:solid 2px blue;}
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <div>
        <div>
            <div>冒泡型事件</div>
        </div>
    </div>
</div>
</div>
</body>
</html>

```

//向闭包内传递变量 i 的值

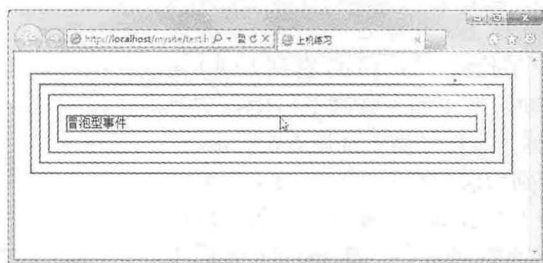


图 5.7 事件冒泡动画演示效果

【注意】

实际上事件冒泡仅是事件流的一种类型，它还包括捕获型事件流，演示示意图如图 5.8 所示。从 DOM 标准角度分析，事件流一般可以分为捕捉阶段、目标阶段和冒泡阶段 3 个阶段。标准事件流一般都会包括这 3 个阶段，演示示意图如图 5.9 所示。



图 5.8 捕捉型事件流示意图

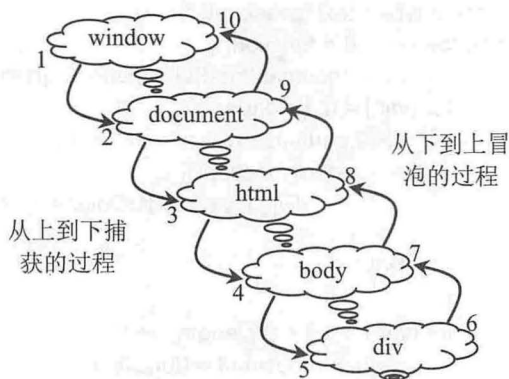


图 5.9 标准事件流示意图

不同类型的浏览器在响应事件时，因为事件流的类型不同，在复杂程序设计中往往会呈现不同的设计效果。同时，对于不同类型的浏览器及其版本，事件流所能影响的 DOM 树顶不同。

- ☑ IE 5.5 及其以下版本：div→body→document。
- ☑ IE 6.0 及其以上版本：div→body→html→document。
- ☑ Mozilla 1.0 及其以上版本：div→body→html→document→window。

5.2.3 事件控制

不管是什么类型的浏览器，都会为 Event 对象预定操纵事件流的方法。例如，在 DOM 标准事件模型中，使用 Event 对象的 stopPropagation() 方法可以中止事件继续向上级层次传播；而对于 IE 浏览器来说，则可以使用 Event 对象的 cancelBubble 属性来阻止，只要设置该属性值为 true 即可。有关事件流的更详细控制操作将在后面章节中进行详细讲解。

一些事件往往被预定了特定的动作，也就是说它包含特定的语义。例如，当单击 a 元素时会自动进行导航；而单击提交按钮时，会自动提交表单；单击重设按钮时，会自动清除表单域的输入值等。如果要控制这些特定的动作，可以在这些事件处理函数中返回 false 即可，从而取消事件的默认动作。例如，在下面示例中，当单击超链接之后，超链接默认的动作将失效，仅弹出一个提示对话框，显示超链接地址。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
window.onload = function(){
    var a = document.getElementsByTagName("a")[0];
    a.onclick = function(){
        alert(a.getAttribute("href"));
        return false;
    }
}
</script>
<title>上机练习</title>
<a href="http://www.baidu.com/">百度一下</a>
</body>
</html>
```

5.3 jQuery 事件封装机制

为了能够更好地兼容不同类型的浏览器，jQuery 在 JavaScript 基础上，进一步封装了不同类型的事件模型，从而形成一种功能更强大、用法更优雅的 jQuery 事件模型。jQuery 事件模型体现如下特征：

- ☑ 统一了事件处理中的各种方法。
- ☑ 允许在每个元素上为每个事件类型建立多个处理程序。
- ☑ 采用 2 级事件模型中标准的事件类型名称。
- ☑ 统一了 Event 对象的传递方法，并对 Event 对象的常用属性和方法进行规范。
- ☑ 为事件管理和操作提供统一的方法。

考虑到 IE 浏览器不支持事件流中的捕获型阶段，且开发者很少使用捕获阶段，jQuery 事件模型也没有

支持事件流中的捕获型阶段。除了这一点区别外，jQuery 事件模型的功能与 2 级事件模型基本相似。

5.3.1 注册事件

1. bind()方法

jQuery 定义了 bind()方法作为统一的接口，用来为每一个匹配元素绑定事件处理程序。具体用法如下：

bind(eventType,[eventData],handler(eventObject))

bind(eventType,[eventData],false)

bind(events)

- ☑ 参数 eventType 表示一个包含一个或多个 JavaScript 事件类型的字符串，如 click、submit，也可以是自定义事件的名称。
- ☑ 参数 eventData 表示将要传递给事件处理函数的数据映射。
- ☑ 参数 handler(eventObject)表示每当事件触发时执行的函数，该参数函数的参数 eventObject 表示传递的事件对象。
- ☑ 参数 false 表示将第 3 个参数设置为 false，用来绑定一个函数，防止默认事件，阻止事件冒泡。
- ☑ 参数 events 表示一个或多个 JavaScript 事件函数，表示要传递的将被执行的动作。

【示例 9】分别为文档中的 p 元素绑定单击事件，这样当单击段落文本时会提示显示当前段落文本。

演示效果如图 5.10 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").bind("click",function(){
        alert($(this).text());
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
</body>
</html>
```



图 5.10 使用 bind()方法为段落文本绑定 click 事件

如果希望向事件处理函数传递更多的信息,则可以把这些信息封装在一个对象结构中,然后把这个对象作为 bind()方法的第2个参数,从而实现事件外与事件内之间进行数据通信。

【示例 10】 在示例 9 的基础上向其传递两个值 A 和 B,则先使用对象结构对其进行封装,然后作为参数传递给 bind()方法。在事件处理函数中可以通过 Event 对象的数据属性来访问这个对象,进而访问该对象内包含的数据。演示效果如图 5.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").bind("click",{a:"A",b:"B"},function(event){
        $(this).text(event.data.a + event.data.b);
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
</body>
</html>
```



图 5.11 在 bind()方法中传递附加数据

【提示】

如果既想取消元素特定事件类型的默认行为,又想阻止事件起泡,可以设置事件处理函数返回值为 false。代码如下:

```
$(“p”).bind(“click”,{a:“A”,b:“B”},function(event){
    $(this).text(event.data.a + event.data.b);
    return false;
});
```

也可以使用 preventDefault()方法取消默认的行为,代码如下:

```
$(“p”).bind(“click”,{a:“A”,b:“B”},function(event){
    $(this).text(event.data.a + event.data.b);
    event.preventDefault();
});
```

或者使用 `stopPropagation()` 方法阻止事件起泡，代码如下：

```
$( "p" ).bind( "click", { a: "A", b: "B" }, function( event ) {
    $( this ).text( event.data.a + event.data.b );
    event.stopPropagation();
});
```

2. 事件方法

除了 `bind()` 方法外，jQuery 还定义了 20 个快捷方法为特定的事件类型绑定事件处理程序，这些方法与 2 级事件模型中的事件类型一一对应，名称完全相同，如表 5.2 所示。

表 5.2 绑定特定事件类型的方法

<code>blur()</code>	<code>focus()</code>	<code>mousedown()</code>	<code>resize()</code>
<code>change()</code>	<code>keydown()</code>	<code>mousemove()</code>	<code>scroll()</code>
<code>click()</code>	<code>keypress()</code>	<code>mouseout()</code>	<code>select()</code>
<code>dblclick()</code>	<code>keyup()</code>	<code>mouseover()</code>	<code>submit()</code>
<code>error()</code>	<code>load()</code>	<code>mouseup()</code>	<code>unload()</code>

例如，对于下面使用 `bind()` 方法绑定的事件：

```
$( "p" ).bind( "click", function() {
    alert( $( this ).text() );
});
```

可以直接使用 `click()` 方法绑定：

```
$( "p" ).click( function() {
    alert( $( this ).text() );
});
```

注意，当使用这些快捷方法时，无法向 `event.data` 属性传递额外的数据。如果不为这些方法传递事件处理函数而直接调用它们，则会触发已绑定这些对象上的对应事件，包括默认的动作。

3. one() 方法

`one()` 方法是 `bind()` 方法的一个特例，由它绑定的事件在执行一次响应之后就会失效。具体用法如下：

`one(eventType,[eventData],handler(eventObject))`

- ☑ 参数 `eventType` 表示一个包含一个或多个 JavaScript 事件类型的字符串，如 `click`、`submit`，或者自定义事件的名称。
- ☑ 参数 `eventData` 表示将要传递给事件处理函数的数据映射。
- ☑ 参数 `handler(eventObject)` 表示每当事件触发时执行的函数。

【示例 11】使用 `one()` 方法绑定的鼠标单击事件，这样它只能够响应一次，当第 2 次单击段落文本时就不再响应。演示效果如图 5.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $( "p" ).one( "click", function() {
        alert( $( this ).text() );
    });
});
```



```

))
</script>
<title>上机练习</title>
</head>
<body>
<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
</body>
</html>

```



图 5.12 使用 one() 方法绑定事件

这个方法的设计思路是在事件处理函数内部增加了注销当前事件的代码。

5.3.2 注销事件

jQuery 定义了 `unbind()` 方法，该方法与 `bind()` 方法是反向操作，能够从每一个匹配的元素中删除绑定的事件。如果没有指定参数，则删除所有绑定的事件，包括使用 `bind()` 方法注册的自定义事件。具体用法如下：

```

unbind(eventType, handler(eventObject))
unbind(eventType, false)
nbind(event)

```

- ☑ 参数 `eventType` 表示一个包含一个或多个 JavaScript 事件类型的字符串，如 `click`、`submit`，也可以是自定义事件的名称。
- ☑ 参数 `eventData` 表示将要传递给事件处理函数的数据映射。
- ☑ 参数 `handler(eventObject)` 表示每当事件触发时执行的函数，该参数函数的参数 `eventObject` 表示传递的事件对象。
- ☑ 参数 `false` 表示将第 3 个参数设置为 `false`，用来绑定一个函数，防止默认事件，阻止事件冒泡。
- ☑ 参数 `event` 表示一个或多个 JavaScript 事件函数，表示要传递的将被执行的动作。

【示例 12】 分别为 `p` 元素绑定 `click`、`mouseover`、`mouseout` 和 `dblclick` 事件类型。在 `dblclick` 事件类型的事件处理函数中调用 `unbind()`，这样在没有双击段落文本之前，鼠标的移过、移出和单击都会触发响应。一旦双击段落文本，则所有类型的事件都被注销，鼠标的移过、移出和单击动作就不再有效应。演示效果如图 5.13 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">

```

```

$(function(){
    $("p").dblclick(function(){           //注册双击事件
        $("p").unbind();                 //注册注销所有事件
    });
    $("p").click(f);                     //注册单击事件
    $("p").mouseover(f);                 //注册鼠标移过事件
    $("p").mouseout(f);                 //注册鼠标移出事件
    function f(event){                  //事件处理函数
        this.innerHTML = "事件类型 = " + event.type;
    }
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>

```



图 5.13 绑定多个事件类型

【提示】

如果提供了事件类型作为参数，则只删除该类型的绑定事件。例如，下面代码将只注销 `mouseover` 事件类型，而其他类型的事件依然有效：

```

$("p").dblclick(function(){
    $("p").unbind("mouseover");
});

```

如果把在绑定时传递的处理函数作为第 2 个参数，则只有这个特定的事件处理函数会被删除。

【示例 13】 分别为 `p` 注册鼠标经过事件，并绑定两个事件处理函数，这样当鼠标经过段落文本时，会分别调用这两个事件处理函数。但是单击段落文本时，将移出其中一个事件处理函数，而再次移过段落文本时，将只有一个事件处理函数被调用。演示效果如图 5.14 所示。

```

<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){           //注册单击事件
        $("p").unbind("mouseover", e); //注销鼠标经过事件中 e()事件处理函数
    });
    $("p").mouseover(f);               //注册鼠标经过事件，绑定 f()事件处理函数
    $("p").mouseover(e);               //注册鼠标经过事件，绑定 e()事件处理函数
    function f(){
        $(this).text("第 1 个单击事件")
    }
    function e(){

```

```
$(this).text("第 2 个单击事件")
}
})
</script>

<p>段落文本</p>
```



图 5.14 注销事件

5.4 jQuery 事件应用

当使用 `bind()`、`one()` 或者其他方法注册事件时，`Event` 对象实例将作为第 1 个参数传递给事件处理函数，这与 2 级事件模型是完全相同的。但是 jQuery 统一了 IE 事件模型和 2 级事件模型中 `Event` 对象属性和方法的使用，使其完全符合 DOM 标准事件模型的规范。除了 `Event` 对象的生僻属性外，jQuery 修正了 Web 开发中可能遇到的浏览器兼容性问题，而不再为了浏览器兼容方式而烦恼。如表 5.3 所示是 jQuery 的 `Event` 对象可以安全使用的属性和方法。

表 5.3 Event 对象可以安全使用的属性和方法

属性/方法	说 明
type	获取事件的类型，如 <code>click</code> 、 <code>mouseover</code> 等。返回值为事件类型的名称，该名称与注册事件处理函数时使用的名称相同
target	发生事件的节点。一般利用该属性来获取当前被激活事件的具体对象
relatedTarget	引用与事件的目标节点相关的节点。对于 <code>mouseover</code> 事件来说，它是鼠标移到目标上时所离开的那个节点；对于 <code>mouseout</code> 事件来说，它是离开目标时鼠标将要进入的那个节点
altKey	表示在声明鼠标事件时，是否按下了 <code>Alt</code> 键。如果返回值为 <code>true</code> ，则表示按下
ctrlKey	表示在声明鼠标事件时，是否按下了 <code>Ctrl</code> 键。如果返回值为 <code>true</code> ，则表示按下
shiftKey	表示在声明鼠标事件时，是否按下了 <code>Shift</code> 键。如果返回值为 <code>true</code> ，则表示按下
metaKey	表示在声明鼠标事件时，是否按下了 <code>Meta</code> 键。如果返回值为 <code>true</code> ，则表示按下
which	当在声明 <code>mousedown</code> 、 <code>mouseup</code> 和 <code>click</code> 事件时，显示鼠标键的状态值，也就是说哪个鼠标键改变了状态。返回值为 1，表示按下左键；返回值为 2，表示按下中键；返回值为 3，表示按下右键
keyCode	当在声明和 <code>keypress</code> 事件时，显示触发事件的键盘键的数字编码，不区分大小写， <code>a</code> 和 <code>A</code> 都返回 65。对于 <code>keypress</code> 事件请使用 <code>which</code> 属性，因为 <code>which</code> 属性跨浏览器时依然可靠
pageX	对于鼠标事件来说，指定光标指针相对于页面原点的水平坐标
pageY	对于鼠标事件来说，指定光标指针相对于页面原点的垂直坐标
screenX	对于鼠标事件来说，指定光标指针相对于屏幕原点的水平坐标
screenY	对于鼠标事件来说，指定光标指针相对于屏幕原点的垂直坐标
data	存储事件处理函数第 2 个参数所传递的额外数据
preventDefault()	取消可能引起任何语义操作的事件，如元素特定事件类型的默认动作
stopPropagation()	防止事件沿着 DOM 树向上传播

5.4.1 事件触发

在表单设计中表单域元素都拥有 `focus()` 和 `blur()` 方法，调用它们将会直接调用对应的 `focus` 和 `blur` 事件处理函数，使文本域获取焦点或者失去焦点。jQuery 也模拟了在脚本控制下自动触发事件处理函数的一系列方法，其中最常用的是 `trigger()` 方法。该方法为给定的事件类型执行所有处理器和行为附加到匹配的元素，具体用法如下：

```
trigger(eventType, extraParameters)
trigger(event)
```

- ☑ 参数 `eventType` 表示一个包含一个或多个 JavaScript 事件类型的字符串，如 `click` 或者 `submit`。
- ☑ 参数 `extraParameters` 表示一个额外的参数数组，利用该参数可以向调用的事件处理函数传递额外的数据。
- ☑ 参数 `event` 表示一个 `jQuery.Event` 对象。

【示例 14】本应该在用户单击时才能够触发的事件处理程序，现在利用 `trigger()` 方法，把其定义在鼠标移过事件处理函数中，从而当鼠标移过段落文本时，会自动触发鼠标单击事件。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").click(function(){
        $("p").text("鼠标单击事件");
    });
    $("p").mouseover(function(){
        $("p").trigger("click"); //调用 trigger()方法直接触发 click 事件
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>
```

【提示】

`trigger()` 方法也会触发同名的浏览器默认行为。例如，如果用 `trigger()` 触发一个 `submit` 事件类型，则同样会导致浏览器提交表单。如果要阻止这种默认行为，则可以在事件处理函数中设置返回值为 `false`。

所有触发的事件都会冒泡到 DOM 树顶。例如，如果在 `p` 元素上触发一个事件，它首先会在这个元素上触发，然后向上冒泡，直到触发 `Document` 对象。这个事件对象有一个 `target` 属性指向最开始触发这个事件的元素。读者可以用 `stopPropagation()` 方法来阻止事件冒泡，或者在事件处理函数中返回 `false` 即可。

`triggerHandler()` 方法对 `trigger()` 方法进行补充，该方法的行为表现与 `trigger()` 方法类似，用法也相同：

```
triggerHandler(eventType, extraParameters)
```

但是 `triggerHandler()` 方法与 `trigger()` 方法存在以下 3 个主要区别：

- ☑ `triggerHandler()` 方法不会触发浏览器默认事件。

- ☑ triggerHandler()方法只触发 jQuery 对象集合中第 1 个元素的事件处理函数。
- ☑ triggerHandler()方法返回的是事件处理函数的返回值,而不是 jQuery 对象。如果最开始的 jQuery 对象集合为空,则这个方法返回 undefined。

除了 trigger()和 triggerHandler()方法外, jQuery 还为大部分事件类型提供了快捷触发的方法,如表 5.4 所示。

表 5.4 jQuery 定义的快捷触发事件的方法

blur()	dblclick()	keydown()	select()
change()	error()	keypress()	submit()
click()	focus()	keyup()	

这些方法没有参数,直接引用能够自动触发引用元素绑定的对应事件处理程序。

【示例 15】针对示例 14,也可以直接使用 click()方法替代 trigger("click")方法。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){
        $("p").text("鼠标单击事件");
    });
    $("p").mouseover(function(){
        $("p").click();    //调用 click()方法快速触发 click 事件
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>
```

5.4.2 事件切换

事件切换在 Web 开发中经常会用到,如样式交互、行为交互等。前面曾经介绍过 toggleClass()方法,该方法就能够实现显示/隐藏样式类,即实现样式动态切换。jQuery 定义了两个事件切换的合成方法: hover()和 toggle(),它们能够实现行为动态交互效果。

1. toggle()方法

toggle()方法能够为 click 事件类型绑定两个事件处理函数,并确保每次点击后依次调用不同的函数。它与直接为 click 事件绑定两个函数的功能不同。具体用法如下:

```
toggle(handler(eventObject), handler(eventObject), [ handler(eventObject)])
```

- ☑ 参数 handler(eventObject)表示第 1 次(奇数)点击时要执行的函数。
- ☑ 参数 handler(eventObject)表示第 2 次(偶数)点击时要执行的函数。

☑ 参数 `handler(eventObject)` 是一个可选参数，表示更多次点击时要执行的函数。

`toggle()` 方法可以包含多个函数参数。如果点击了一个匹配的元素，则触发指定的第 1 个函数；当再次点击同一元素时，则触发指定的第 2 个函数；如果有更多函数，则再次触发，直到最后一个。随后的每次点击都重复对这几个函数的轮番调用。

【示例 16】为 `input` 元素注册了一个 `toggle` 合成事件，这样每次单击按钮时，将会循环轮流调用参数中指定的事件处理函数。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").toggle(
        function(){
            this.value = "第 1 次单击";
        },
        function(){
            this.value = "第 2 次单击";
        },
        function(){
            this.value = "第 3 次单击";
        }
    )
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="持续单击" />
</body>
</html>
```

`toggle()` 方法比 `toggleClass()` 方法更加实用，它可以根据元素被点击的次数切换元素的启用状态。例如，切换显示元素的不透明度。对于 `toggleClass()` 方法来说，可以使用 `unbind("click")` 来删除它。

2. hover()方法

`hover()` 方法可以模仿悬停事件，即鼠标移动到一个对象上面，以及移出这个对象的事件交替触发的方法。这是一个自定义的方法，它将两个事件函数绑定到匹配元素上，当鼠标指针进入和离开元素时被分别执行。具体用法如下：

`hover(handlerIn(eventObject), handlerOut(eventObject))`

☑ 参数 `handlerIn(eventObject)` 表示当鼠标指针进入元素时触发执行的事件函数。

☑ 参数 `handlerOut(eventObject)` 表示当鼠标指针离开元素时触发执行的事件函数。

【示例 17】为按钮绑定 `hover` 合成事件，这样当鼠标移过按钮时，会触发指定的第 1 个函数；当鼠标移出这个元素时，会触发指定的第 2 个函数。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
```



```

<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("input").hover(
        function(){
            this.value = "鼠标经过";
        },
        function(){
            this.value = "鼠标已移出";
        }
    )
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="鼠标切换事件" />
</body>
</html>

```

【提示】

在 JavaScript 中, mouseout 事件存在一个很严重的错误, 如果鼠标移到当前元素包含的子元素上时, 将会触发当前元素的 mouseout 和 mouseover 事件。

【示例 18】 为 div 元素绑定 mouseover 和 mouseout 事件处理程序, 当鼠标进入 div 元素时将会触发 mouseover 事件, 而当鼠标移到 span 元素上时, 虽然鼠标并没有离开 div 元素, 但是将会触发 mouseout 和 mouseover 事件。如果鼠标在 div 元素内部移动, 就可能会不断触发 mouseout 和 mouseover 事件, 产生不断闪烁的事件触发现象。演示效果如图 5.15 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var p = document.getElementsByTagName("p")[0];
    var span = document.getElementsByTagName("span")[0];
    if(div.addEventListener){
        //兼容非 IE
        div.addEventListener("mouseover",over,false); //注册 mouseover 事件
        div.addEventListener("mouseout",out,false); //注册 mouseout 事件
    }else{
        //兼容 IE
        div.attachEvent("onmouseover",over); //注册 mouseover 事件
        div.attachEvent("onmouseout",out); //注册 mouseout 事件
    }
    function over(event){
        //事件处理函数
        var event = event || window.event; //兼容 Event 对象
        p.innerHTML += event.type + "<br />";
    }
    function out(event){
        //事件处理函数
        var event = event || window.event; //兼容 Event 对象
        p.innerHTML += event.type + "<br />";
    }
}

```

```

}
</script>
<style type="text/css">
div { width:300px; height:180px; background:red; padding:20px; }
span { float:right; width:120px; height:80px; background:blue; color:white; font-weight:bold; }
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <span></span>
</div>
<p></p>
</body>
</html>

```

在 jQuery 中, 由于 bind()、mouseover() 和 mouseout() 方法都是直接在原事件基础上进行包装的, 因此使用 jQuery 的 bind()、mouseover() 和 mouseout() 方法绑定时也会存在上述问题。

不过, hover() 方法修正了这个错误, 它会伴随着对鼠标是否仍然处在特定元素中的检测。如果是, 则会继续保持悬停状态, 而不触发移出事件。

【示例 19】 针对示例 18, 使用 hover() 来实现相同的设计效果, 当鼠标进入 div 元素, 并在 div 元素内部移动时, 只会触发一次 mouseover 事件。演示效果如图 5.16 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").hover(                                //绑定 hover()合成事件
        function(event){                          //注册 mouseover 事件处理函数
            $("p").append(event.type + "<br />");
        },
        function(event){                          //注册 mouseout 事件处理函数
            $("p").append(event.type + "<br />");
        }
    )
})
</script>
<style type="text/css">
div { width:300px; height:180px; background:red; padding:20px; }
span { float:right; width:120px; height:80px; background:blue; color:white; font-weight:bold; }
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <span></span>
</div>
<p></p>

```



```
</body>
</html>
```

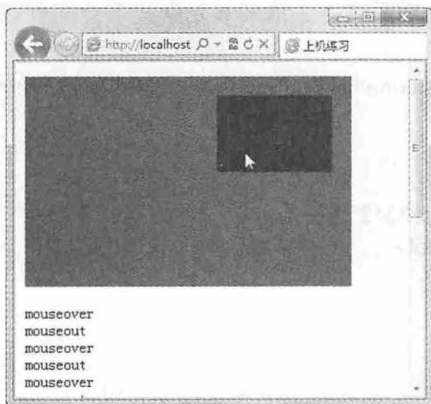


图 5.15 mouseout 事件响应

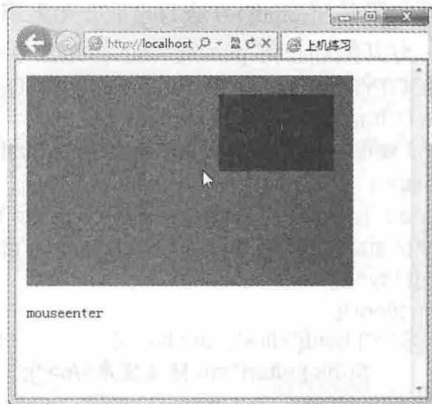


图 5.16 hover()方法响应

5.4.3 事件委派

委派是一种方法引用的类型，一旦为委派分配了方法，委派将与该方法具有完全相同的行为。委派方法的使用可以像其他任何方法一样，具有参数和返回值。为此 jQuery 定义了事件委派的方法 `live()`。`live()` 方法的用法与 `bind()` 方法相同，第 1 个参数用于设置事件类型，第 2 个参数用于设置事件处理函数，具体用法如下：

```
live(eventType, handler)
```

```
live(eventType, eventData, handler)
```

- ☑ 参数 `eventType` 表示一个包含一个 JavaScript 事件类型的字符串，如 `click`、`keydown`。在 jQuery 1.4 中，该字符串可以包含多个用空格分隔的事件类型或自定义事件名称。
- ☑ 参数 `handler` 表示每次事件触发时会执行的函数。
- ☑ 参数 `eventData` 表示将要传递给事件处理函数的数据映射。

【示例 20】为 `p` 元素委派一个 `click` 事件，这样在后面动态生成的 `p` 元素也会拥有这个 `click` 事件。因此，单击页面中任何一个存在的 `p` 元素，都会调用事先委派的事件处理函数。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").live("click", function(){ //委派事件
        $(this).after("<p>段落文本</p>");
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
```



```
</body>
</html>
```

但是如果使用 `bind()` 方法为当前 `p` 元素绑定 `click` 事件, 则在后面动态生成的 `p` 元素就不会拥有这个 `click` 事件, 只有开始存在的 `p` 元素绑定了 `click` 事件。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("p").bind("click", function(){ //绑定事件
        $(this).after("<p>段落文本</p>");
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>
```

`live()` 方法与 `bind()` 方法还有一个区别, 那就是 `live()` 方法一次只能绑定一个事件, 而 `bind()` 方法可以绑定多个事件。

【注意】

jQuery 的 `live()` 方法仅支持 `click`、`dblclick`、`mousedown`、`mouseup`、`mousemove`、`mouseover`、`mouseout`、`keydown`、`keypress` 和 `keyup` 事件类型的委派, 不支持 `blur`、`focus`、`mouseenter`、`mouseleave`、`change` 和 `submit` 等事件类型。目前 `live` 事件只能支持使用选择器选择的元素, 如 `$("li a").live(...)`, 但是不支持类似于 `$("a", someElement).live(...)` 或者 `$("a").parent().live(...)`。

`live` 事件冒泡的行为与传统的方式不同, 因此也不能完全支持 `stopPropagation()` 或者 `stopImmediatePropagation()` 阻止冒泡, 但部分支持。如果内外元素都用 `live` 事件绑定, 则可以通过 `return false` 来阻止冒泡。如果外部父元素是普通事件, 而内部子元素是 `live` 事件, 则无法通过 `return false` 来阻止冒泡。

【拓展】

如果要移除 `live()` 绑定的事件, 可以用 `die()` 方法实现。具体用法如下:

```
die()
```

```
die(eventType, [handler])
```

☑ 参数 `eventType` 表示一个包含一个或多个 JavaScript 事件类型的字符串, 如 `click`、`keydown`, 或自定义事件的名称。

☑ 参数 `handler` 表示不再执行的函数, 该方法返回 jQuery 对象。

任何通过 `live()` 方法附加的事件处理函数都可以使用 `die()` 删除, 这个方法类似于调用不带参数的 `unbind()` 方法, 用来删除先前用 `bind()` 绑定的所有事件。

【示例 21】 当单击段落文本后, 会自动解除事件委派, 则第 2 行文本就不再拥有鼠标单击事件, 但是第 1 个 `div` 元素依然保持单击事件。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").live("click", function(){    //委派事件
        $(this).after("<p>段落文本</p>");
        $("p").die("click");           //解除委派
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本</p>
</body>
</html>

```

【提示】

die()方法与 live()方法是两个相反的操作。如果 die()方法不带参数,则所有绑定的 live 事件都会被移除。如果设置 type 参数,那么会移除对应的 live 事件。如果同时指定了第 2 个参数,则只移出指定的事件处理函数。

5.4.4 事件命名空间

jQuery 支持事件命名空间,以方便对事件进行管理。所谓事件命名空间,就是在事件类型后面以点语法附加一个别名,以便引用事件,如 click.a,其中 a 就是 click 当前事件类型的别名,即事件命名空间。

【示例 22】 为 div 元素绑定多个事件类型,然后使用命名空间进行规范,从而方便管理。

```

<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click.a", function(){    //绑定 click 事件
        $("body").append("<p>click 事件</p>");
    });
    $("div").bind("dblclick.a", function(){    //绑定 dblclick 事件
        $("body").append("<p>dblclick 事件</p>");
    });
    $("div").bind("mouseover.a", function(){    //绑定 mouseover 事件
        $("body").append("<p>mouseover 事件</p>");
    });
    $("div").bind("mouseout.a", function(){    //绑定 mouseout 事件
        $("body").append("<p>mouseout 事件</p>");
    });
})

```

```

</script>
<title>上机练习</title>
</head>
<body>
<div>jQuery 事件命名空间</div>
</body>
</html>

```

在所绑定的事件类型后面附加命名空间，这样在删除事件时，就可以直接指定命名空间即可。例如，调用下面一行代码就可以把示例 22 中绑定的事件全部删除：

```
$("#div").unbind(".a");
```

同样，要为相同的事件类型设置不同的命名空间，如果仅删除某一个事件处理程序，则只需要指定命名空间即可。

【示例 23】 如果直接单击段落文本，会触发命名空间为 a 的 click 事件和命名空间为 b 的 click 事件，当单击按钮之后，则删除命名空间为 a 的事件类型，再次单击段落文本，就只能触发命名空间为 a 的 click 事件。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("#div").bind("click.a", function(){
        $("#body").append("<p>click.a 事件</p>");
    });
    $("#div").bind("click.b", function(){
        $("#body").append("<p>click.b 事件</p>");
    });
    $("#input").click(function(){
        $("#div").unbind(".a"); //注销命名空间为 a 的事件
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<div>jQuery 命名空间</div>
<input type="button" value="删除事件" />
</body>
</html>

```

另外，在 trigger() 方法中，如果事件类型后面附加感叹号，则表示触发不包含命名空间的特定事件类型。例如，在下面示例中，当单击按钮时，将会触发没有命名空间的 click 事件。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >

```



```

$(function(){
    $("div").bind("click", function(){
        $("body").append("<p>click 事件</p>");
    });
    $("div").bind("click.b", function(){
        $("body").append("<p>click.b 事件</p>");
    });
    $("input").click(function(){
        $("div").trigger("click!");    //注意 click 类型后面的感叹号
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<div>jQuery 命名空间</div>
<input type="button" value="删除事件" />
</body>
</html>

```

5.4.5 绑定多个事件

jQuery 提供了多种方法为同一个对象绑定多个事件，这些方法适用不同的开发环境以及习惯用法，以方便设计师加快开发速度。

【示例 24】 为当前 div 元素绑定了两个 click 事件，当单击 div 元素时，分别会触发这个绑定的事件处理函数。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click", function(){    //绑定 click 事件 1
        $("body").append("<p>click 事件 1</p>");
    });
    $("div").bind("click", function(){    //绑定 click 事件 2
        $("body").append("<p>click.b 事件 2</p>");
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<div>绑定多个事件</div>
</body>
</html>

```

对于代码也可以以链式语法把它们给串在一起，代码如下：

```

$(function(){
    $("div").bind("click", function(){

```

```

        $("body").append("<p>click 事件 1</p>");
    }).bind("click", function(){
        $("body").append("<p>click 事件 2</p>");
    });
}

```

使用 bind() 方法可以为元素一次绑定多个事件类型。

【示例 25】 在同一个 bind() 方法中同时绑定了 mouseover 和 mouseout 事件类型。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("div").bind("mouseover mouseout", function(event){    //同时绑定多个事件类型
        $("body").append(event.type + "<br />");
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<div>绑定多个事件</div>
</body>
</html>

```

在示例 25 中，当光标移过 div 元素时，会触发 mouseover 事件，调用绑定的事件处理函数，而当光标移出 div 元素时，再次触发 mouseout 事件，再次调用该函数。上面代码也可以这样来写：

```

$(function(){
    $("div").bind("mouseover", function(event){                //绑定 mouseover 事件
        $("body").append(event.type + "<br />");
    });
    $("div").bind("mouseout", function(event){                //绑定 mouseout 事件
        $("body").append(event.type + "<br />");
    });
})

```

5.4.6 自定义事件

jQuery 支持自定义事件，所有自定义事件都可以通过 jQuery() 函数触发。

【示例 26】 自定义一个 delay 事件类型，并把它绑定到 input 元素对象上，然后在按钮单击事件中触发自定义事件，以实现延迟响应的设计效果。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){

```

```

    $("input").bind("delay", function(event){ //自定义并绑定 delay 事件类型
        setTimeout(function(){ //延迟响应
            alert(event.type);
        },1000);
    });
    $("input").click(function(){ //绑定 click 事件
        $("input").trigger("delay"); //触发自定义事件
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="jQuery 自定义事件" />
</body>
</html>

```

实际上，自定义事件就是自定义函数，触发自定义事件就相当于调用自定义函数。由于自定义事件拥有事件类型的很多特性，因此在开发中拥有特殊的用途。

5.4.7 页面初始化事件

jQuery 定义了 `ready()` 方法用于封装 JavaScript 原生的 `window.onload` 方法。`ready()` 方法表示当 DOM 载入就绪，并可以查询和被操纵时，能够自动执行的函数。它是 jQuery 事件模型中最最重要的一个函数，极大地提高了 Web 应用程序的响应速度。具体用法如下：

```
ready(handler)
```

参数 `handler` 表示当 DOM 完全加载完成时执行一个函数。下面几种写法都是等价的：

```
$(document).ready(handler)
```

```
$(document).ready(handler) (this is not recommended)
```

```
$(handler)
```

```
$(document).bind("ready", handler)
```

注意，如果 `ready` 事件已准备触发，再尝试用 `bind("ready")` 绑定处理函数将不会被执行。`ready()` 方法通常用于一个匿名函数：

```

$(document).ready(function() {
    handler
});

```

如果 `ready()` 在 DOM 被初始化后被调用，新的处理函数 `handler` 将立即执行。

【提示】

虽然 JavaScript 提供了 `load` 事件，当页面呈现时执行这个事件，直到所有的东西，如图像已被完全接收前，此事件不会被触发。在大多数情况下，只要 DOM 结构已完全加载，脚本就可以运行。传递给处理函数 `ready()` 能保证 DOM 准备好后就执行这个函数，所以附加所有其他的事件处理程序和运行其他 jQuery 代码，这个方法通常是最好的地方。当使用脚本依赖 CSS 样式属性值时，重要的是要引用外部样式或引用样式元素放置在脚本之前。

如遇到有代码依赖于加载元素，例如，如果一个图像的尺寸为必须项，代码应该被放置在该处理函数中，用来替换 `load` 事件。

`ready()` 方法通常是不符合 `<body onload="">` 属性的。如果 `load` 必须使用，要么使用 `ready()`，要么使用 jQuery 的 `load()` 方法来附加 `load` 事件处理函数到 `window` 或更多的具体对象，如图片。

【示例 27】 分别为 3 个 div 元素绑定 ready 事件，在浏览器中预览，则可以看到绑定的 3 个事件都在文档加载完毕后被集中触发。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$("div").eq(0).ready(function(){
    alert(1);
});
$("div").eq(1).ready(function(){
    alert(2);
});
$("div").eq(2).ready(function(){
    alert(3);
});
</script>
<title>上机练习</title>
</head>
<body>
<div>模块 1</div>
<div>模块 2</div>
<div>模块 3</div>
</body>
</html>
```

【分解】

jQuery 的 ready 事件与 JavaScript 的 load 事件具有相同的功能，但是它们在触发时机方面还存在细微区别：

- ☑ JavaScript 的 load 事件是在文档内容完全加载完毕后才被触发。这个文档内容包括页面中所有节点以及节点关联的文件，这时 JavaScript 才可以访问网页中的任何元素和内容。这种情况对于编写功能性的代码非常有利，因为无须考虑加载的次序。
- ☑ jQuery 的 ready 事件是在 DOM 完全就绪时被触发的，此时文档中所有元素都是可以访问的，但是与文档关联的文件可能还没有下载完毕。通俗地说，就是浏览器下载并完成解析 HTML 的 DOM 树结构，代码就可以运行。

例如，对于一个大型图库网站来说，为页面中所有显示的图像绑定一个初始化设置的脚本。如果使用 JavaScript 原生的 load 事件来设计，那么用户在使用这个页面之前，必须等待页面中所有图像下载完毕才能够实现。而在 load 事件等待图像加载过程中，如果行为还未添加到那些已经加载的图像上，则此时如果用户操作它们，可能会导致很多意想不到的尴尬。而使用 jQuery 的 ready 事件，则在 DOM 树结构解析之后，就立即触发页面初始化事件，从而避免使用 load 事件所带来的尴尬。

但是，由于 jQuery 的 ready 事件过早地触发，虽然 DOM 树结构已经解析完毕，但是很多元素的属性未必生效。例如，很多图像还没有加载完毕，导致这些图像的属性无效，如图像的高度和宽度。要解决这个问题，可以使用 jQuery 的 load 事件进行触发，该事件等效于 JavaScript 的 load 事件。

另外，JavaScript 的 load 事件存在一个很严重的缺陷，就是它不允许多次调用。例如，在下面示例中分两次调用 load 事件，但是当网页加载完毕后，JavaScript 仅触发了第 2 个 load 事件调用。

```
window.onload = function(){
    alert("一次调用 load 事件");
};
```

```
}  
window.onload = function(){  
    alert("二次调用 load 事件");  
}
```

实际上,第1次事件调用已经被第2个调用所覆盖。要解决两次调用之间的冲突问题,可以把两个页面初始化函数放在同一个load事件中。例如,针对上面示例可以按如下方式进行修改:

```
window.onload = function(){  
    (function(){  
        alert("一次调用 load 事件");  
    })();  
    (function(){  
        alert("二次调用 load 事件");  
    })()  
}
```

上面代码直接在load事件的处理函数中定义和调用两个匿名函数。当然也可以把这两个匿名函数改为函数声明的方式定义,然后在load事件处理函数中调用。

通过间接的方式解决load事件多次调用问题,但是load事件仍然存在很多局限。例如,在多个JavaScript文件中,可能每个JavaScript文件都会用到window.load()方法,在这种情况下使用上面的方法是无法解决的,同时无法保证按顺序执行多个注册的函数。

而jQuery的ready事件能够很好地解决这个问题,在同一个文档中可以进行多次调用。例如,针对上面示例,可以使用如下方法轻松解决,即便在不同JavaScript文件中,都可以无限制地多次调用ready事件:

```
$(function(){  
    alert("一次调用 load 事件")  
});  
$(function(){  
    alert("二次调用 load 事件")  
});
```

第 6 章

Ajax 应用

( 视频讲解: 1 小时 12 分钟)

Ajax 是 Asynchronous JavaScript and XML 的缩写, 即异步 JavaScript 和 XML。实际上 Ajax 并非缩写词, 而是由 Jesse James Gaiett 创造的名词, 它是指一种创建交互式网页应用的网页开发技术。在基于数据的应用中, 用户需求的数据, 如用户信息、联系人列表, 都可以从独立于实际网页的服务端取得, 并且可以被动态地写入网页中, 给缓慢的 Web 应用体验着色, 使之像桌面应用一样流畅。

Ajax 的核心是 JavaScript 对象 XMLHttpRequest。该对象在 IE 5 中首次引入, 它是一种支持异步请求的技术。简而言之, XMLHttpRequest 能够帮助用户使用 JavaScript 向服务器提出请求并处理响应, 而不阻塞网页交互响应。考虑到浏览器的兼容性, jQuery 封装了 Ajax 应用, 避免了用户为了适应不同浏览器的规则而编写大量的代码。

6.1 XMLHttpRequest 基础

XMLHttpRequest 对象是 Ajax 异步通信的技术核心, 各主流浏览器通过组件嵌入的方式提供支持。要深入理解 Ajax, 应该先熟悉 XMLHttpRequest 对象, 了解该对象包含的方法、属性以及它们的基本用法。在此基础上再理解 jQuery 封装的 Ajax 应用就会轻松许多。

6.1.1 XMLHttpRequest 对象

XMLHttpRequest 可以提供不重新加载页面的情况下更新网页, 在页面加载后在客户端向服务器请求数据, 在页面加载后在服务器端接收数据, 在后台向客户端发送数据。XMLHttpRequest 对象提供了对 HTTP 协议的完全访问, 包括 POST 请求、HEAD 请求以及普通的 GET 请求的能力。XMLHttpRequest 可以同步或异步返回 Web 服务器的响应, 并且能以文本或者一个 DOM 文档形式返回内容。尽管名为 XMLHttpRequest, 它并不限于与 XML 文档一起使用, 它可以接收任何形式的文本文档。XMLHttpRequest 对象是名为 Ajax 的 Web 应用程序架构的一项关键功能。

XMLHttpRequest 得到了所有现代浏览器较好的支持。唯一的浏览器依赖性涉及 XMLHttpRequest 对象的创建。在 IE 5 和 IE 6 中, 必须使用特定于 IE 的 ActiveXObject() 构造函数。

XMLHttpRequest 对象还没有标准化, 但是 W3C 已经开始了标准化的工作, 当前的 XMLHttpRequest 实现已经相当一致, 但是和标准有细微的不同。例如, 一个实现可能返回 null, 而标准要求是空字符串, 或者实现可能把 readyState 设置为 3, 而不保证所有的响应头部都可用。

XMLHttpRequest 对象共包含 8 个基本属性和 6 个基本方法，这些属性和方法负责完成异步通信的全部工作，说明如表 6.1 和表 6.2 所示。

表 6.1 XMLHttpRequest 对象的属性

属 性	说 明
onreadystatechange	指定当 readyState 属性改变时的事件处理句柄
readyState	返回当前请求的状态
status	返回当前请求的 HTTP 状态码
statusText	返回当前请求的响应行状态
responseBody	返回正文信息
responseStream	以文本流的形式返回响应信息
responseText	以字符串的形式返回响应信息
responseXML	以 XML 数据的形式返回响应信息

表 6.2 XMLHttpRequest 对象的方法

方 法	说 明
open()	创建一个新的 HTTP 请求，并指定此请求的方法、URL 以及验证信息（用户名/密码）
send()	发送请求到 HTTP 服务器并接收回应
getAllResponseHeaders()	获取响应的所有 HTTP 头信息
getResponseHeader()	从响应信息中获取指定的 HTTP 头信息
setRequestHeader()	单独指定请求的某个 HTTP 头信息
abort()	取消当前请求

使用 XMLHttpRequest 对象实现异步通信一般需要下面几个步骤：

- (1) 定义 XMLHttpRequest 对象实例。
- (2) 调用 open() 方法建立与服务端端的连接。
- (3) 注册 onreadystatechange 事件处理函数，以便接收和处理从服务器端响应的信息。
- (4) 调用 send() 方法发送请求。

6.1.2 实例化 XMLHttpRequest

直接使用 JavaScript 实现异步通信，则应该先定义 XMLHttpRequest 对象，由于不同浏览器定义 XMLHttpRequest 对象的方式不同，因此必须考虑浏览器兼容性问题。定义 XMLHttpRequest 对象的详细代码如下：

```
<script type="text/javascript">
//定义 XMLHttpRequest 对象
if (window.XMLHttpRequest){           //兼容 Mozilla、Safari 等非 IE 浏览器
    var xmlhttprequest = new XMLHttpRequest();
}
else if (window.ActiveXObject){       //兼容 IE 浏览器
    try{
        var xmlhttprequest = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e){
        try{
            xmlhttprequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
}
```

```

    }
    catch (e)
    {}
  }
}
</script>

```

在上面代码中，使用条件语句分别判断当前浏览器的类型，并根据不同类型浏览器决定定义对象的方法。对于非 IE 浏览器来说，一般都支持 window.XMLHttpRequest 对象，可以使用 new XMLHttpRequest() 方法定义 XMLHttpRequest 对象。由于 IE 浏览器支持 ActiveXObject 组件实现 Ajax 技术，只有使用 new ActiveXObject("Msxml2.XMLHTTP") 或 new ActiveXObject("Microsoft.XMLHTTP") 方法定义 XMLHttpRequest 对象。

6.1.3 建立连接

XMLHttpRequest 对象包含很多方法和属性，虽然不同浏览器的定义方式不同，但是它们都包含相同的方法和属性，且用法和参数基本相似，这为开发人员提供了便利。

定义 XMLHttpRequest 对象之后，调用 open() 方法可以建立异步连接。具体用法如下：

```
xmlhttprequest.open(Method, Url, Async, User, Password)
```

该方法包含 5 个参数，其中前两个参数是必需的：

- ☑ xmlhttprequest 表示 XMLHttpRequest 对象实例。
- ☑ 参数 Method 表示 HTTP 方法，如 POST、GET、PUT 和 PROPFIND，方法的名称不区分大小写。
- ☑ 参数 Url 表示请求的地址。可以是绝对地址，也可以是相对地址。
- ☑ 参数 Async 为可选项，设置是否为异步通信，默认为 true，表示可以异步；而取值为 false 时，表示必须同步通信。
- ☑ 参数 User 和 Password 表示请求的文件需要服务器进行验证，如果未指定，当服务器需要验证时，会弹出验证窗口要求进行验证。

例如：

```
xmlhttprequest.open("GET","http://localhost/mysite/test.asp", false);
```

在上面的代码中，利用 open() 方法为 xmlhttprequest 对象建立了一个与 test.asp 文件的连接。此时 open() 方法包含以下 3 个参数：

- ☑ 第 1 个参数设置发送 HTTP 请求为 GET 方法，即以查询字符串的形式传输数据（把数据附加在 URL 后面）。如果要上传大量数据，则应该使用 POST 方法。
- ☑ 第 2 个参数用来设置要打开的服务器文件（该文件可以是任意类型的文件），这里要打开的是服务器端 mysite 站点内 server.asp 文件。
- ☑ 第 3 个参数用来指定不要异步请求。所谓异步就是发出请求之后，不需要等待服务器端的响应，用户可以继续执行其他操作。默认为 true，表示异步请求。

【示例 1】 先调用 open() 方法打开一个请求，然后调用 send() 方法发送请求的信息。如果服务器响应了请求，则会把响应信息发送给 XMLHttpRequest 对象的 responseText 属性。最后，直接读取 XMLHttpRequest 对象的 responseText 属性值即可。

在服务器端的 test.txt 文件中包含这样一行信息“Hi,How are you!”。然后在客户端访问本案例文件，在网页中单击“向服务器发出异步请求”按钮，则会获取服务器端响应的信息，如图 6.1 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>

```

```

<script type="text/javascript" >
//定义 XMLHttpRequest 对象
if (window.XMLHttpRequest){                                //兼容 Mozilla、Safari 等非 IE 浏览器
    var xmlhttprequest = new XMLHttpRequest();
}
else if (window.ActiveXObject){                            //兼容 IE 浏览器
    try{
        var xmlhttprequest = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e){
        try{
            xmlhttprequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch (e)
        {}
    }
}
window.onload = function(){                                //页面初始化处理函数
    var input = document.getElementsByTagName("input")[0]; //获取按钮
    input.onclick = function(){                            //绑定 click 事件类型
        xmlhttprequest.open("GET","test.txt", false);      //调用 XMLHttpRequest 对象的 open()方法, 打开与服务
                                                            务器之间的同步通信连接
        xmlhttprequest.send(null);                          //向服务器发送请求
        alert("来自服务器端的问候: " + xmlhttprequest.responseText); //显示服务器的影响信息
    }
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="向服务器发出异步请求" />
</body>
</html>

```



图 6.1 异步请求演示效果

6.1.4 请求和响应

6.1.3 节的示例已经演示了如何发送请求以及如何接收服务器端响应信息的一般方法。实际上, 当建立连接之后, 用户就可以使用 `send()` 方法发送请求到 HTTP 服务器端, 并接收服务器的响应。具体用法如下:

```
xmlhttprequest.send(varBody)
```


参数 `varBody` 表示欲通过此请求发送的数据。该参数可以传递客户端发送给服务器端的请求数据。如果不传递信息,则可设置参数值为 `null`。

`send()`方法虽然能够接收服务器端的响应,但是如何把接收的数据读取出来,还需用到 `responseBody`、`responseStream`、`responseText` 或者 `responseXML` 属性来接收异步响应的信息,这些属性的描述信息请参阅表 6.2 的介绍。

【示例 2】 演示如何在异步交互中向服务器端传递简单的字符串信息,以及如何在服务器端使用服务器端技术接收这些传递的信息,并把这些信息响应给客户端,最后由客户端回调函数捕获并显示出来。演示效果如图 6.2 所示。



图 6.2 GET 请求和响应

注意,本示例以及后面章节异步请求示例的服务器端技术都是以 ASP 技术为基础的,请读者在上机练习之前先在本系统构建虚拟的 IIS 运行环境。

整个案例的实现步骤如下:

(1) 由于 ASP 虚拟服务器与 Windows 操作系统捆绑在一起,所以不需要安装任何软件,只需要在系统控制面板中启动管理工具,然后打开 Internet 信息服务(IIS)管理器即可,在其中定义虚拟站点,或者对 IIS 服务器进行设置。

提示,如果读者在控制面板的工具管理窗口中没有看到 IIS 管理器,则可以单击拆卸程序按钮,在打开的 Windows 组件管理选项中安装 IIS 服务组件即可。

(2) 当 IIS 组件包安装好之后,就可以把 ASP 文件放置在 `inetpub\wwwroot` 目录下,然后在浏览器地址栏中输入“`http://localhost/`”地址即可。最后在相应的文件夹中选定页面文件,即可运行 Ajax 应用示例。

(3) 新建 HTML 页面,保存为 `test.html`,在页面中输入以下脚本。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
//定义 XMLHttpRequest 对象
if (window.XMLHttpRequest){
    //兼容 Mozilla、Safari 等非 IE 浏览器
    var xmlhttprequest = new XMLHttpRequest();
}
else if (window.ActiveXObject){
    //兼容 IE 浏览器
    try{
        var xmlhttprequest = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e){
        try{
            xmlhttprequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
}
```

```

    }
    catch (e)
    {}
  }
}
window.onload = function(){
  var input = document.getElementsByTagName("input")[0];
  input.onclick = function(){
    xmlhttprequest.open("GET","test.asp?name= zhangsan ", false); //建立连接
    xmlhttprequest.send(null);          //发送请求
    alert(xmlhttprequest.responseText); //提示服务器端响应信息
  }
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="向服务器发出异步请求" />
</body>
</html>

```

在上面脚本中，通过在 `open()` 方法的第 2 个 URL 参数值末尾附加查询字符串的形式，向服务器端传递请求的数据。

(4) 新建服务器端脚本处理文件，使用该文件接收并处理用户响应信息，保存为 `test.asp`，在该文件中输入下面代码。

```

<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<%
var name = Request.QueryString("name");
if(name && name == " zhangsan ") {          //如果存在该参数，且参数值等于 zhangsan
  Response.Write(name + "是合法的用户名。");
}
else{                                         //否则提示其他信息
  Response.Write(name + "非法的用户名");
}
%>

```

上面脚本是以 JavaScript 脚本形式进行书写的，`Request` 和 `Response` 作为 ASP 的两个基本对象而存在，它们分别用来接收请求信息和发送响应信息。

首先，使用 `Request` 对象的 `QueryString` 集合对象读取请求的 URL 中的查询字符串信息。如果存在 `name` 参数值，且该参数值等于 `zhangsan`，则响应给客户端一种提示信息，否则就响应另一种信息。响应信息通过 `Response` 对象的 `write()` 方法实现。

然后，客户端可以借助 `XMLHttpRequest` 对象的 `responseText` 属性读取服务器端响应信息的文本字符串，最后把这些信息以提示的形式显示出来。

(5) 在客户端打开 IE 浏览器，然后在地址栏中输入“`http://localhost/mysite/test.html`”，按 Enter 键访问 `test.html` 页面，然后在页面中单击“向服务器发出异步请求”按钮，则会接收到服务器端响应的信息，如图 6.2 所示。

6.2 jQuery Ajax

jQuery 对 Ajax 操作进行了封装，在 jQuery 中，`ajax()` 方法属于最底层的方法，第 2 层是 `load()`、`get()`、

post()方法,第3层是 getScript()和 getJSON()方法。有关 jQuery 封装 Ajax 的所有方法及其说明如表 6.3 所示。

表 6.3 jQuery Ajax 方法及其说明

方 法	说 明
ajax()	执行一个异步的 HTTP (Ajax) 的请求
ajaxComplete()	当 Ajax 请求完成后注册一个回调函数,这是一个 Ajax 事件
ajaxError()	Ajax 请求出错时注册一个回调处理函数,这是一个 Ajax 事件
ajaxSend()	每当一个 Ajax 请求即将发送时注册一个回调处理函数,这是一个 Ajax 事件
ajaxSetup()	设置未来(可以理解为全局)的 Ajax 请求默认选项
ajaxStart()	在 Ajax 请求刚开始时执行一个处理函数,这是一个 Ajax 事件
ajaxStop()	在 Ajax 请求停止后隐藏加载信息
ajaxSuccess()	当一个 Ajax 请求成功完成时显示一个信息
get()	通过服务器 HTTP GET 请求加载数据
getJSON()	通过 HTTP GET 请求从服务器载入 JSON 数据
getScript()	通过 HTTP GET 请求从服务器载入并执行一个 JavaScript 文件
load()	载入远程 HTML 文件代码并插入至 DOM 中
param()	创建一个序列化的数组或对象,适用于一个 URL 地址查询字符串或 Ajax 请求
post()	通过服务器 HTTP POST 请求加载数据
serialize()	将用作提交的表单元素的值编译成字符串
serializeArray()	将用作提交的表单元素的值编译成拥有 name 和 value 对象组成的数组,如[{name: a value: 1}, { name: b value: 2 },...]

6.2.1 设计一个简单的示例

在系统、深入学习 Ajax 应用之前,先看一个简单的示例。设想在客户端单击按钮之后,向服务器端发出一个异步请求,然后把服务器端的响应显示出来。

要实现这样的设想,需要先在虚拟服务器端建立一个文本文件(test.txt),在这个文件中输入下面的字符串:

```
Hello World
```

然后新建一个静态页面,保存为 test.html,在客户端的静态网页文件中输入下面的 jQuery 代码:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.get("test.txt", function(data){
            alert("来自服务器端的问候: " + data);
        });
    });
})
</script>
<title>上机练习</title>
```



```

</head>
<body>
<input type="button" value="向服务器发出异步请求" />
</body>
</html>

```

打开网页浏览器，在地址栏中输入该文件的地址，然后按 Enter 键确认访问该页面。注意，mysite 表示本地定义的虚拟服务器名称。

`http://localhost/mysite/test.html`

在该静态页面中单击“向服务器发出异步请求”按钮，则会显示如图 6.3 所示的提示信息。



图 6.3 GET 请求和响应

【分解】

在这个示例中，jQuery 的公共方法 `get()` 能够以异步方式向服务器发出一个简单的请求，然后把服务器端的响应信息存储在 `get()` 方法的回调函数参数中，从而完成一次异步通信的过程。在客户端通过读取回调函数的参数，就可以获取服务器端的响应信息。

`get()` 方法可以包含 4 个参数，其中第 1 个参数表示请求的服务器端地址；第 2 个参数及其后面所有参数都是可选参数，其中第 2 个参数可以是请求的信息；第 3 个参数表示回调函数；第 4 个参数表示响应信息的类型。在上面示例中，简化了 `get()` 方法的使用，仅传递了服务器端的 URL 以及回调函数参数。

6.2.2 GET 请求

`get()` 方法就是通过查询字符串的方式来传递请求信息。GET 请求的参数通过问号 (?) 前缀附加在 URL 的末尾，参数是以连字符 (&) 连接的一个或多个名/值对。每个名称和值都必须在编码后才能用在 URL 中，可以在 JavaScript 中使用 `encodeURIComponent()` 方法进行编码，服务器端在接收这些数据时也必须使用 `decodeURIComponent()` 方法进行解码。

当使用 `XMLHttpRequest` 对象发送一个 GET 请求时，只需将包含所有参数的 URL 传入 `open()` 方法，同时设置第 1 个参数值为 GET。这样服务器就能够自动在 URL 后面的查询字符串中接收到客户端传递过来的信息。使用 GET 请求比较简单，也非常方便。URL 最大长度为 2048 字符 (2KB)，它适合传递一些简单的参数信息，不易传输大容量或受保护的数据。

在 GET 请求中可以发送更多的参数信息。例如，在下面连接请求中，共传递了 3 个参数值：参数变量 `name` 的值等于 `zhangsan`，参数变量 `pass` 的值等于 `123456`，参数变量 `age` 的值等于 `1`。

```
xmlhttprequest.open("GET","test1.asp?name=zhangsan&pass=123456&age=1",false); //建立连接
```

jQuery 定义了 `get()` 方法，专门负责通过远程 HTTP GET 请求方式载入信息。该方法是一个简单的 GET 请求功能，以取代复杂的 `$.ajax()` 方法。该方法的具体用法如下：

```
jQuery.get( url, [data], [success(data, textStatus, jqXHR)], [dataType])
```

`get()` 方法包含 4 个参数，其中第 1 个参数为必须设置项，后面 3 个参数为可选参数：

☑ 参数 `url` 表示要请求页面的 URL 地址。

- ☑ 参数 data 表示一个对象结构的名/值对列表。
- ☑ 参数 success(data, textStatus, jqXHR)表示异步交互成功之后调用的回调函数。回调函数的参数值为服务器端响应的信息。
- ☑ 参数 dataType 表示服务器端响应信息返回的内容格式，如 XML、HTML、Script、JSON 和 Text，或者_default。

【示例 3】使用 get()方法向服务器端的 test.asp 文件发出一个请求，并把一组数据传递给该文件，然后在回调函数中读取并显示服务器端响应的信息。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){           //绑定 click 事件
        $.get("test.asp",{                 //向 test1.asp 文件发出请求
            name : "zhangsan",             //发送的请求信息
            pass : 123456,
            age : 1
        },function(data){                 //回调函数
            alert(data);                   //显示响应信息
        });
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="向服务器发出异步请求" />
</body>
</html>
```

【提示】

get()方法能够在请求成功时调用回调函数。如果需要在出错时执行函数，则必须使用\$.ajax()方法。可以把 get()方法的第 2 个参数所传递的数据以查询字符串的形式附加在第 1 个参数 URL 后面。例如，针对上面 get()方法的用法，还可以这样书写：

```
$.get("test1.asp?name=zhangsan&pass=123456&age=1",function(data){ //回调函数
    alert(data);                                                    //显示响应信息
});
```

jQuery 还定义了两个专用方法 getJSON()和 getScript()。这两个方法的功能和用法与 get()是完全相同的，不过 getJSON()方法能够请求载入 JSON 数据，getScript()方法能够请求载入 JavaScript 文件。

这两个方法与 get()方法的用法基本相同，但是仅支持 get()方法的前 3 个参数，不需要设置第 4 个参数，即指定响应数据的类型，因为方法本身已经说明了接收的信息类型。例如，在服务器端文件（test5.asp）中输入下面响应信息：

```
[
    {name:"zhu",pass:"123456",age:"1"},
    {name:"zhang",pass:"abcdef",age:"2"},
    {name:"zhao",pass:"opqrst",age:"3"}
]
```

上面信息以 JSON 格式进行编写, 整个数据包含在一个数组中, 每个数组元素是一个对象, 对象中包含 3 个属性, 分别是 name、pass 和 age。

【示例 4】在客户端的 jQuery 脚本中, 使用 `getJSON()` 方法请求服务器端文件 (`test1.asp`), 并把响应信息解析为数据表格形式显示。演示效果如图 6.4 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.getJSON("test1.asp",function(data){           //使用 getJSON()方法发送请求并接收 JSON 格式数据
            var data = data;                             //获取响应数据
            var str = "<table border=1 width=100%>";      //定义字符串临时变量
            str += "<tr>";
            for(var name in data[0]){                     //遍历响应数据中的第 1 个数组元素对象
                str += "<th>" + name + "</th>";             //获取并显示元素对象的属性名
            }
            str += "</tr>";
            for(var i=0; i<data.length; i++){             //遍历响应数据中的数组元素
                str += "<tr>";
                for(var name in data[i]){                 //遍历数组元素中的每个属性成员
                    str += "<td>" + data[i][name] + "</td>"; //获取并显示元素对象的属性值
                }
                str += "</tr>";
            }
            str += "<table>";
            $("div").html(str);                           //把临时字符串以 HTML 格式嵌入到 div 元素中显示
        });
    });
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="向服务器发出异步请求" />
<div></div>
</body>
</html>
```



图 6.4 使用 `getJSON()` 方法获取并解析 JSON 格式数据

使用 `getScript()` 方法能够异步请求并导入外部 JavaScript 文件，具体示例不再演示。

6.2.3 POST 请求

POST 请求方式与 GET 请求方式截然不同。POST 请求支持发送任意格式、任意长度的数据，而不仅仅限于名/值对字符串。传递二进制的文件、大容量信息、安全信息或 XML 格式数据时，使用 POST 方式比较高效。

一般来说，POST 请求用于在表单中输入数据后的提交过程。与 GET 请求相似，POST 请求的参数也必须进行编码，并用连字符（&）进行分隔。这些参数在发送 POST 请求时，不会被附加到 URL 的末尾，而是作为 `send()` 方法的参数进行传递，然后被送到服务器端。

例如，针对 `test1.asp?name=zhangsan&pass=123456&age=1` 查询字符串，可以作为参数传递给 `send()` 方法，代码如下：

```
send("name=zhangsan&pass=123456&age=1");
```

如果使用 POST 方式模仿 6.2.1 节示例效果实现把 `name=zhangsan` 参数信息传递给服务器，则可以使用如下代码：

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("POST","test2.asp", false); //建立连接
        xmlhttprequest.setRequestHeader('Content-type','application/x-www-form-urlencoded');
        xmlhttprequest.send("name=zhangsan");           //发送请求
        alert(xmlhttprequest.responseText);             //提示服务器端响应信息
    }
}
</script>
```

```
<input type="button" value="向服务器发出异步请求" />
```

使用 POST 方式进行请求，需要对部分代码进行修改：

☑ 在 `open()` 方法中设置第 1 个参数为 POST，表明当前请求连接是 POST 方式。

☑ 调用 `setRequestHeader()` 方法，设置请求的消息头，指定请求内容类型为 `application/x-www-form-urlencoded`。`application/x-www-form-urlencoded` 类型表示传递的是表单值，一般使用 POST 发送请求时都必须设置该选项，否则服务器会无法识别传递过来的数据。`setRequestHeader()` 方法的具体用法如下：

```
xmlhttprequest.setRequestHeader("Header-name", "value")
```

为了方便服务器能够识别当前请求为 Ajax 异步请求，一般设置头部信息中 `User-Agent` 首部为 `XMLHTTP`，以便于服务器端能够辨别出 `XMLHttpRequest` 异步请求和其他客户端普通请求。例如：

```
xmlhttprequest.setRequestHeader("User-Agent", "XMLHTTP");
```

这样就可以在服务器端编写脚本分别为现代浏览器和不支持 JavaScript 的浏览器呈现不同的文档，以提高可访问性的手段。如果使用 POST 方法传递数据，就必须设置另一个头部信息：

```
xmlhttprequest.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
```

用于发送 POST 请求的数据类型（Content Type）通常是 `application/x-www-form-urlencoded`，这意味着还可以以 `text/xml` 或 `application/xml` 类型给服务器直接发送 XML 数据，甚至以 `application/json` 类型发送 JavaScript 对象数据。例如，下面的示例将向服务器端发送 XML 类型的数据，而不是简单的名/值对参数：

```
xmlhttprequest.send("<bookstore><book id='1'>书名</book>< /bookstore >")
```

读者可以访问 <http://www.w3.org/Protocols/HTTP/HTTRQ-Headers.html> 了解 HTTP 请求头信息的总览表。

☑ 在 `send()` 方法中传递参数值, 该值是一个或多个名/值对, 多个名/值对之间使用 “&” 分隔符进行分隔。这样在 ASP 服务器端就可以利用 `Request.Form()` 方法捕获所传递过来的值。

在名/值对中, “名” 可以为表单域的名称 (与表单域相对应), “值” 可以是固定的值, 也可以是一个变量。如果是变量, 可以把表单域内包含的值直接传递给变量, 再由变量负责把数据传递给服务器端。

☑ 必须把 `open()` 方法中的第 3 个参数值设置为 `false`, 即关闭异步通信。如果不需要通过 `send()` 方法传递数据, 则只要传递 `null` 作为参数值即可。

最后, 还需要对服务器端的请求文件进行修改, 使用 `Request.Form()` 方式获取客户端传递的参数值。

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
```

```
<%
```

```
var name = Request.Form("name");
```

```
if(name && name == "zhangsan"){           //如果存在该参数, 且参数值等于 zhangsan
```

```
    Response.Write(name + "是合法的用户名。");
```

```
}
```

```
else{                                     //否则提示其他信息
```

```
    Response.Write(name + "非法的用户名");
```

```
}
```

```
%>
```

注意, 针对 GET 和 POST 方式, 服务器端获取数据的方法也是不同的, 不同的服务器技术可能会略有区别, 上面示例主要根据 ASP 技术进行演示。

jQuery 定义了 `post()` 方法, 专门负责通过远程 HTTP POST 请求方式载入信息。该方法是一个简单的 POST 请求功能, 以取代复杂的 `$.ajax()` 方法。

`post()` 方法包含 4 个参数, 与 `get()` 方法相似。其中, 第 1 个参数为必须设置的参数, 后面 3 个参数为可选参数:

☑ 第 1 个参数表示要请求页面的 URL 地址。

☑ 第 2 个参数表示一个对象结构的名/值对列表。

☑ 第 3 个参数表示异步交互成功之后调用的回调函数。回调函数的参数值为服务器端响应的信息。

☑ 第 4 个参数表示服务器端响应信息返回的内容格式, 如 XML、HTML、Script、JSON 和 Text, 或者 `_default`。

例如, 在下面这个示例中, 使用 `post()` 方法向服务器端的 `test2.asp` 文件发出一个请求, 并把一组数据传递给该文件, 然后在回调函数中读取并显示服务器端响应的信息。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
```

```
<script type="text/javascript">
```

```
$(function(){
```

```
    $("input").click(function(){           //绑定 click 事件
```

```
        $.post("test2.asp",{               //向 test2.asp 文件发出请求
```

```
            name : "zhangsan",             //发送的请求信息
```

```
            pass : 123456,
```

```
            age : 1
```

```
        },function(data){                 //回调函数
```

```
            alert(data);                   //显示响应信息
```

```
    });
```

```
});
```

```
)
```

```
</script>
```

```
<input type="button" value="jQuery 实现的异步请求" />
```

通过上面示例可以看到 `post()` 方法与 `get()` 方法在用法上是完全相同的, 数据传递和接收响应信息的方式

都相同，唯一区别是请求方式不同。具体选用哪个方法，主要取决于客户端所要传递的数据容量和格式，同时应该考虑服务器端接收数据的处理方式。

不管是 `get()` 方法，还是 `post()` 方法，它们都是一种简单的请求方式，对于特殊的数据请求和响应处理，应该选择 `$.ajax()` 方法。`ajax()` 方法的参数比较多且复杂，能够处理各类特殊的异步交互行为，关于这个问题请参阅 6.2.4 节内容。

6.2.4 ajax()方法请求

`ajax()` 方法是 jQuery 实现 Ajax 的底层方法，也就是说它是 `get()`、`post()` 等方法的基础，使用该方法可以完成通过 HTTP 请求加载远程数据。由于 `ajax()` 方法的参数较为复杂，在没有特殊需求时，使用高级方法（如 `get()`、`post()` 等）即可。

`ajax()` 方法只有一个参数，即一个列表结构的对象，包含各配置及回调函数信息。例如，加载 JavaScript 文件，则可以使用下面的参数选项：

```
$.ajax({  
    type: "GET",           //请求方式  
    url: "test.js",       //请求文件的 URL  
    dataType: "script"    //响应的数据类型  
});
```

如果把客户端的数据传递给服务器端，并获取服务器的响应信息，则可以使用类似下面的参数选项：

```
$.ajax({  
    type: "POST",          //请求方式  
    url: "test.asp",      //请求文件的 URL  
    data: "name=John&location=Boston", //传递给服务器的数据  
    success: function(data){ //异步通信成功后的回调函数  
        alert(data);        //显示服务器的响应信息  
    }  
});
```

加载 HTML 页面，则可以使用下面的参数选项：

```
$.ajax({  
    url: "test.html",      //请求文件的 URL  
    cache: false,         //禁止缓存  
    success: function(html){ //异步通信成功后的回调函数  
        $("#box").append(html); //把 HTML 片段附加到当前文档的盒子中  
    }  
});
```

如果希望以同步方式加载数据，则可以使用下面的选项设置。使用同步方式加载数据时，其他用户操作将被锁定。

```
var html = $.ajax({  
    url: "test.asp",      //请求文件的 URL  
    async: false          //同步请求  
});
```

`ajax()` 方法参数选项列表如表 6.4 所示。

表 6.4 ajax()方法参数选项列表

参 数	数 据 类 型	说 明
async	Boolean	设置是否异步请求。默认为 true，即所有请求均为异步请求。如果需要发送同步请求，设置为 false 即可。注意，同步请求将锁住浏览器，用户其他操作必须等待请求完成才可以执行

参 数	数 据 类 型	说 明
beforeSend	Function	发送请求前可修改 XMLHttpRequest 对象的函数，如添加自定义 HTTP 头。XMLHttpRequest 对象是唯一的参数。 该函数如果返回 false，可以取消本次 Ajax 请求
cache	Boolean	设置缓存。默认值为 true，dataType 为 script 时，默认为 false。设置为 false 将不会从浏览器缓存中加载请求信息
complete	Function	请求完成后回调函数（请求成功或失败时均调用）。该函数包含两个参数：XMLHttpRequest 对象和一个描述成功请求类型的字符串
contentType	String	发送信息至服务器时的内容编码类型。默认为 application/x-www-form-urlencoded
data	Object、String	发送到服务器的数据。将自动转换为请求字符串格式，必须为 Key/Value 格式。GET 请求中将附加在 URL 后。查看 processData 选项说明以禁止此自动转换。如果为数组，jQuery 将自动为不同值对应同一个名称。如 {foo:["bar1", "bar2"]} 转换为 '&foo=bar1&foo=bar2'
dataFilter	Function	给 Ajax 返回的原始数据进行预处理的函数。提供 data 和 type 两个参数：data 是 Ajax 返回的原始数据，type 是调用 jQuery.ajax 时提供的 dataType 参数。函数返回的值将由 jQuery 进一步处理
dataType	String	预期服务器返回的数据类型。如果不指定，jQuery 自动根据 HTTP 包含的 MIME 信息返回 responseXML 或 responseText，并作为回调函数参数传递，可用值如下。 <input checked="" type="checkbox"/> xml: 返回 XML 文档，可用 jQuery 处理 <input checked="" type="checkbox"/> html: 返回纯文本 HTML 信息，包含的 script 标签会在插入 DOM 时执行 <input checked="" type="checkbox"/> script: 返回纯文本 JavaScript 代码。不会自动缓存结果，除非设置了 cache 参数。注意，在远程请求时（不在同一个域下），所有 POST 请求都将转为 GET 请求（因为将使用 DOM 的 script 标签来加载） <input checked="" type="checkbox"/> json: 返回 JSON 数据 <input checked="" type="checkbox"/> jsonp: JSONP 格式。使用 JSONP 形式调用函数时，如 myurl?callback=？，jQuery 将自动替换“？”为正确的函数名，以执行回调函数 <input checked="" type="checkbox"/> text: 返回纯文本字符串
error	Function	请求失败时调用函数。该函数包含 3 个参数：XMLHttpRequest 对象、错误信息、（可选）捕获的错误对象。如果发生了错误，错误信息（第 2 个参数）除了得到 null 之外，还可能是 timeout、error、notmodified 和 parsererror
global	Boolean	是否触发全局 Ajax 事件，默认值为 true。设置为 false 将不会触发全局 Ajax 事件，如 ajaxStart 或 ajaxStop 可用于控制不同的 Ajax 事件
ifModified	Boolean	仅在服务器数据改变时获取新数据，默认值为 false。使用 HTTP 包含的 Last-Modified 头信息进行判断
jsonp	String	在一个 jsonp 请求中重写回调函数的名字。这个值用来替代在“callback=？”这种 GET 或 POST 请求中 URL 参数里的 callback 部分，如 {jsonp:'onJsonPLoad'} 会导致将“onJsonPLoad=？”传给服务器
password	String	用于响应 HTTP 访问认证请求的密码
processData	Boolean	发送的数据将被转换为对象（技术上讲并非字符串）以配合默认内容类型 application/x-www-form-urlencoded。默认值为 true，如果要发送 DOM 树信息或其他不希望转换的信息，则设置为 false
scriptCharset	String	只有当请求时 dataType 为 jsonp 或 script，并且 type 是 GET 才会用于强制修改 charset。通常在本地和远程的内容编码不同时使用

续表

参 数	数 据 类 型	说 明
success	Function	请求成功后的回调函数。函数的参数由服务器返回，并根据 dataType 参数进行处理后的数据；描述状态的字符串
timeout	Number	设置请求超时时间（毫秒）。此设置将覆盖全局设置
type	String	设置请求方式，如 POST 或 GET，默认为 GET。其他 HTTP 请求方法，如 PUT 和 DELETE 也可以使用，但仅部分浏览器支持
url	String	发送请求的地址，默认为当前页面地址
username	String	用于响应 HTTP 访问认证请求的用户名
xhr	Function	需要返回一个 XMLHttpRequest 对象。默认在 IE 下是 ActiveXObject，而其他情况下是 XMLHttpRequest。用于重写或者提供一个增强的 XMLHttpRequest 对象

如果设置了 dataType 选项，应确保服务器返回正确的 MIME 信息，例如，XML 返回 text/xml。如果设置 dataType 为 script，则在请求时，如果请求文件与当前文件不在同一个域名中，所有 POST 请求都被转换为 GET 请求，因为 jQuery 将使用 DOM 的 script 标签来加载响应信息。

6.2.5 响应状态

异步通信实际上就是不需要刷新的暗部通信方式，这种方式虽然存在很多优势，但是却无法让人直观了解通信的过程和状态。为了避免客户端被动的等待，XMLHttpRequest 对象定义了 readyState 属性，该属性可以动态跟踪异步通信的状态。readyState 属性值说明如表 6.5 所示。

表 6.5 readyState 属性值列表

属 性 值	说 明
0	未初始化。表示对象已经建立，但是尚未初始化，尚未调用 open() 方法
1	初始化。表示对象已经建立，尚未调用 send() 方法
2	发送数据。表示 send() 方法已经调用，但是当前的状态及 HTTP 头未知
3	数据传送中。已经接收部分数据，因为响应及 HTTP 头不全，这时通过 responseBody 和 responseText 获取部分数据会出现错误
4	完成。数据接收完毕，此时可以通过 responseBody 和 responseText 获取完整的响应数据

在异步通信过程中，一旦请求和响应的状态发生变化，也就是说当 readyState 属性值发生变化时，就会触发 onreadystatechange 事件处理函数。

readystatechange 是一种特殊的事件类型，它与 HTTP 传输紧密相关联。每当 HTTP 请求状态改变时，服务器都会回调客户端的 onreadystatechange 事件处理函数。如果 readyState 属性返回值为 4，则说明异步请求和响应完毕，那么就可以放心地读取返回的数据了。

虽然通过 readyState 属性可以确定响应是否完成，但是还存在另一个问题：如果服务器完成响应请求，但是报告了一个错误，此时如果读取响应信息，也容易发生错误。为了防止此类意外，还应坚持 HTTP 通信状态。只有当状态码为 200 时，才表示服务器正确完成响应处理。在 XMLHttpRequest 对象中可以借助 status 属性获取当前的 HTTP 状态码。

例如，定义一个回调函数，检测服务器是否已正确完成响应，然后决定是否读取响应信息。实现的代码如下：

```
function response(){
    if(xmlhttprequest.readyState == 4){
```

```

        if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
            alert(xmlhttprequest.responseText);
        }
    }
}

```

然后，绑定 onreadystatechange 事件处理函数（以 6.2.3 节示例为基础进行演示操作）：

```

<script type="text/javascript">
//省略定义 XMLHttpRequest 对象
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("POST","test2.asp", false);    //建立连接
        xmlhttprequest.setRequestHeader('Content-type','application/x-www-form-urlencoded');
        xmlhttprequest.onreadystatechange = response;    //绑定状态变化事件处理函数
        xmlhttprequest.send("name=zhangsan");    //发送请求
    }
}
function response(){
    ...//省略代码，请参阅上面演示代码
}
</script>

```

```
<input type="button" value="向服务器发出异步请求" />
```

readystatechange 事件类型不会创建事件对象 Event，所以不要在事件处理函数中传递 Event 对象。另外，onreadystatechange 属性应该放置在调用 send() 方法之前。也就是说，在 XMLHttpRequest 对象发送请求之前必须绑定 onreadystatechange 事件处理函数。

jQuery 在 XMLHttpRequest 对象定义的 readyState 属性基础上，对异步交互中服务器响应状态进行封装，提供了 6 个响应事件，以便进一步细化对整个请求响应过程的跟踪，说明如表 6.6 所示。

表 6.6 jQuery 封装的响应状态事件

事 件	说 明
ajaxStart	Ajax 请求开始时进行响应
ajaxSend	Ajax 请求发送前进行响应
ajaxComplete	Ajax 请求完成时进行响应
ajaxSuccess	Ajax 请求成功时进行响应
ajaxStop	Ajax 请求结束时进行响应
ajaxError	Ajax 请求发生错误时进行响应

例如，在下面示例中，为当前异步请求绑定 6 个 jQuery 定义的 Ajax 事件，在浏览器中预览，则可以看到浏览器根据请求和响应的过程，逐步提示过程进展。首先，响应的是 ajaxStart 和 ajaxSend 事件，然后是 ajaxSuccess 事件，最后是 ajaxComplete 和 ajaxStop 事件，如图 6.5 所示。如果请求失败，则中间会响应 ajaxError 事件。

```

<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.ajax({
            type: "POST",

```



```

        url: "test2.asp",
        data: "name=zhangsan"
    });
    $("#div").ajaxStart(function(){
        alert("Ajax 请求开始");
    })
    $("#div").ajaxSend(function(){
        alert("Ajax 请求将要发送");
    })
    $("#div").ajaxComplete(function(){
        alert("Ajax 请求完成");
    })
    $("#div").ajaxSuccess(function(){
        alert("Ajax 请求成功");
    })
    $("#div").ajaxStop(function(){
        alert("Ajax 请求结束");
    })
    $("#div").ajaxError(function(){
        alert("Ajax 请求发生错误");
    })
    });
}
</script>

<input type="button" value="jQuery 实现的异步请求" />

```



图 6.5 jQuery 的 Ajax 事件响应过程

在这些事件中大部分都会包含几个默认参数。例如，`ajaxSuccess`、`ajaxSend` 和 `ajaxComplete` 都包含 `event`、`request` 和 `settings`，其中 `event` 表示事件类型，`request` 表示请求信息，`settings` 表示设置的选项信息。`ajaxError` 事件还包含 4 个默认参数：`event`、`XMLHttpRequest`、`ajaxOptions` 和 `thrownError`。其中，前 3 个参数与上面几个事件方法的参数基本相同，最后一个参数表示抛出的错误。

6.2.6 响应信息

`XMLHttpRequest` 对象定义了 `responseText`、`responseBody`、`responseStream` 和 `responseXML` 属性，分别用来存储服务器端不同的响应格式，具体说明如表 6.7 所示。

表 6.7 获取响应信息方式

属 性	说 明
responseBody	将响应信息正文以 Unsigned Byte 数组形式返回
responseStream	以 ADO Stream 对象的形式返回响应信息
responseText	将响应信息作为字符串返回
responseXML	将响应信息格式化为 XML 文档格式返回

服务器端响应数据一般为文本格式 (responseText) 或 XM 格式 (responseText), 对于二进制数据流可以使用 responseStream 属性存取。获取响应信息之后, 还需要使用 JavaScript 脚本把它们转换为需要的形式进行显示。

XMLHttpRequest 对象允许从服务器端响应任意格式的数据。但在实际应用中, 大多数的 Web 开发人员一般都将格式约定为 XML、HTML、JSON 或其他某种纯文本格式。要使用哪种响应格式, 可以参考下面几条原则:

- ☑ 如果响应 HTML 结构的数据时, 选择 HTML 格式会比较省事, 所要编写的脚本也会很少。此时使用 responseText 属性以字符串格式读取。
- ☑ 如果团队协作开发, 且项目庞杂, 选择 XML 格式会更容易协调, 因为所有成员都比较熟悉和了解这种格式。此时使用 responseXML 属性以 XML 格式读取。
- ☑ 如果检索复杂的响应数据, 且结构复杂, 那么选择 JSON 格式会比较轻松。此时使用 responseText 属性以字符串格式读取。

例如, 下面示例是在 6.2.5 节示例的基础上, 演示如何获取 XML 格式的数据, 并进行解析。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("GET","test3.asp", false);           //建立连接
        xmlhttprequest.onreadystatechange = response;             //绑定状态变化事件处理函数
        xmlhttprequest.send(null);                                 //发送请求
    }
}
function response(){
    if(xmlhttprequest.readyState == 4){
        if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
            var data = xmlhttprequest.responseXML;                //获取 XML 格式的数据
            //利用 DOM 方法解析 XML 数据结构
            var node = data.getElementsByTagName("data")[0];      //获取第 1 个 data 节点
            var text = node.firstChild.data;                      //获取该节点包含的文本节点中的数据
            alert(text);                                           //返回字符串“响应数据”
        }
    }
}
}</script>
```

```
<input type="button" value="异步信息交互" />
```

在服务器端, 设计使用 ASP 脚本生成 XML 文档 (test3.asp):

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<?xml version="1.0" encoding="utf-8"?>
```

```
<%
Response.ContentType = "text/xml"
Response.Write("<data>响应数据</data>")
%>
```

对于 XML 文档结构来说，第 1 行必须是 `<?xml version="1.0" encoding="utf-8"?>`，该行命令表示输出的数据为 XML 格式文档，同时标识了 XML 文档的版本和字符编码。为了能够兼容 IE 和 FF 等浏览器，允许不同浏览器都可以识别 XML 文档，还应该为响应信息定义 XML 文本类型。最后根据 XML 语法规则编写文档的信息结构。当然，也可以直接向服务器端的 XML 格式文件发出请求，此时可以返回该文档的数据。

如果以 JSON 格式解析响应数据，可以按如下方法来设计，示例的完整代码如下：

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("GET","test4.asp", false);           //建立连接
        xmlhttprequest.onreadystatechange = response;             //绑定状态变化事件处理函数
        xmlhttprequest.send(null);                                 //发送请求
    }
}
function response(){
    if(xmlhttprequest.readyState == 4){
        if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
            var data = xmlhttprequest.responseText;               //获取文本格式的数据
            //调用 JavaScript 的 eval()方法进行解析
            var obj = eval("(" + data + ")");                      //把 JavaScript 字符串转换为 JavaScript 对象
            var text = obj.data;                                    //获取对象的属性值
            alert(text);                                           //返回字符串“响应数据”
        }
    }
}
}</script>
```

```
<input type="button" value="异步信息交互" />
```

在服务器端，设计请求的文件（test4.asp）内容如下：

```
{
data:"响应数据"
}
```

6.2.7 载入网页文件

遵循 Ajax 异步交互的设计原则，jQuery 定义了可以加载网页文档的方法 `load()`。该方法与 `getScript()` 方法的功能相似，都是加载外部文件，但是它们的用法完全不同。`load()` 方法能够把加载的网页文件附加到指定的网页标签中。

例如，新建一个简单的网页文件（test.html），代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>静态网页文件</title>
```



```

</head>
<body>
<table width="100%" border="1">
  <tr>
    <th>name</th>
    <th>pass</th>
    <th>age</th>
  </tr>
  <tr>
    <td>zhu</td>
    <td>123</td>
    <td>1</td>
  </tr>
  <tr>
    <td>zhang</td>
    <td>456</td>
    <td>2</td>
  </tr>
  <tr>
    <td>wang</td>
    <td>789</td>
    <td>3</td>
  </tr>
</table>
</body>
</html>

```

然后，在另一个页面中输入下面的 jQuery 脚本：

```

<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
  $("input").click(function(){
    $("div").load("test.html");
  });
})
</script>

<input type="button" value="jQuery 实现的异步请求" />
<div></div>

```

这样当在浏览器中预览时，单击“jQuery 实现的异步请求”按钮后，则会把请求的 test.html 文件中的数据表格加载到当前页面的 div 元素中，如图 6.6 所示。

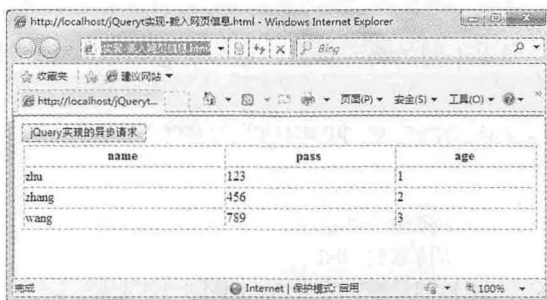


图 6.6 使用 jQuery 的 load() 方法载入外部文件

使用 `ajax()` 方法可以替换 `load()` 方法，因为 `load()` 方法是以 `ajax()` 方法作为底层来实现的。例如，针对上面示例，可以使用下面 jQuery 代码进行替换：

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        var str = ($.ajax({                //调用 ajax()方法，返回 XMLHttpRequest 对象
            url: "test.html",              //载入的 URL
            async: false                    //禁止异步载入
        })).responseText;                 //获取 XMLHttpRequest 对象中包含的服务器响应信息
        $("div").html(str);               //把载入的网页内容附加到 div 元素内
    });
})
</script>

<input type="button" value="jQuery 实现的异步请求" />
<div></div>
```

6.2.8 预设参数项

对于频繁与服务器进行交互的页面来说，每一次交互都要设置很多选项，这种操作是很繁琐的，也容易出错。为此，jQuery 定义了 `ajaxSetup()` 方法，该方法可以预设异步交互中的通用选项，从而减轻频繁设置选项的繁琐。

`ajaxSetup()` 方法的参数仅包含一个参数选项的列表对象，这与 `ajax()` 方法的参数选项设置是相同的。在该方法中设置的选项，可以实现全局共享，从而在具体交互中只需要设置个性化参数即可。

例如，在下面示例中，先使用 `$.ajaxSetup()` 方法把本页面中异步交互的公共选项进行预设，包括请求的服务器端文件、禁止触发全局 Ajax 事件、请求方式、响应数据类型和响应成功之后的回调函数。这样在不同按钮上绑定异步请求时，只需要设置需要发送请求的信息即可。

在服务器端的请求文件（`test6.asp`）中输入下面代码：

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<%
var name = Request.Form("name");
if(name){
    Response.Write("接收到请求信息: " + name);
}
else{
    Response.Write("没有接收到请求信息!");
}
%>
```

这样当单击不同按钮时，会弹出不同的响应信息，这些信息都是从客户端接收到的请求信息，如图 6.7 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $.ajaxSetup({                //预设公共选项
        url: "test6.asp",        //请求的 URL
        global: false,           //禁止触发全局 Ajax 事件
        type: "POST",            //请求方式
        dataType: "text",        //响应数据的类型
    });
})
```

```

        success : function(data){ //响应成功之后的回调函数
            alert(data);
        }
    });
    $("input").eq(0).click(function(){ //为按钮 1 绑定异步请求
        $.ajax({
            data : "name=zhu" //发送请求的信息
        });
    });
    $("input").eq(1).click(function(){ //为按钮 2 绑定异步请求
        $.ajax({
            data : "name=wang" //发送请求的信息
        });
    });
    $("input").eq(2).click(function(){ //为按钮 3 绑定异步请求
        $.ajax({
            data : "name=zhang" //发送请求的信息
        });
    });
    });
</script>

<input type="button" value="异步请求 1" />
<input type="button" value="异步请求 2" />
<input type="button" value="异步请求 3" />

```



图 6.7 接收信息

6.2.9 预处理字符串

在 Ajax 异步通信过程中，客户端所发送的请求字符串格式必须是由“&”字符连接的多个名/值对，如“user=zhu&sex=man&grade=2”。而当使用表单发送请求时，发送请求的信息并非按此格式进行传递。用户需要手工编写发送信息的字符串格式，为了减轻开发人员不必要的劳动量，特意定义了 `serialize()` 方法，该方法能够帮助用户按名/值对的字符串格式快速整理，并返回合法的请求字符串。

例如，在下面这个复杂表单中，用户需要传递的表单值是比较多的，如果一项一项获取并组织为请求字符串，就稍显繁琐。

```

<form action="#" method="post">
    姓名: <input type="text" name="user" /><br />
    性别:

```



```

<input type="radio" name="sex" value="man" checked="checked" />男
<input type="radio" name="sex" value="men" />女<br />
年级:
<select name="grade">
  <option value="1">一</option>
  <option value="2">二</option>
  <option value="3">三</option>
</select><br />
科目:
<select name="kemu" size="6" multiple="multiple">
  <option value="yuwen">语文</option>
  <option value="shuxue">数学</option>
  <option value="waiyu">外语</option>
  <option value="wuli">物理</option>
  <option value="huaxue">化学</option>
  <option value="jisuanji">计算机</option>
</select><br />
兴趣:
<input type="checkbox" name="love" value="yundong" />运动
<input type="checkbox" name="love" value="wenyi" />文艺
<input type="checkbox" name="love" value="yinyue" />音乐
<input type="checkbox" name="love" value="meishu" />美术
<input type="checkbox" name="love" value="youxi" />游戏<br />
<input type="submit" value="提交" id="submit" />
</form>

```

如果在发送请求之前，调用 `serialize()` 方法，就可以轻松解决合法格式的请求字符串的设计。实现代码如下：

```

<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#submit").click(function(){
        $("p").html($("#form").serialize());    //获取和格式化表单的请求字符串信息，并显示出来
        return false;                          //禁止提交表单
    });
})
</script>

```

在浏览器中预览，然后单击“提交”按钮，则可以看到规整的请求字符串，如图 6.8 所示。



图 6.8 预处理请求的字符串

除了 `serialize()` 方法外, jQuery 还定义了 `serializeArray()` 方法, 该方法能够返回指定表单域的值的 JSON 结构的对象。注意, 该方法返回的是 JSON 对象, 而非 JSON 字符串。JSON 对象是由一个对象数组组成的, 其中每个对象包含一个或两个名值对: `name` 参数和 `value` 参数 (如果 `value` 不为空)。

例如, 针对上面的表单结构, 可以设计如下 jQuery 代码, 获取用户传递的请求值, 并把这个 JSON 结构的对象解析为 HTML 字符串显示出来, 如图 6.9 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("#submit").click(function(){
        //var array = $("form").serializeArray();           //注意, 不能够直接在 form 元素上调用该方法
        var array = $("input, select, :radio").serializeArray(); //在表单域对象上调用 serializeArray()方法, 返回
        包含传递表单域和值的 JSON 对象
        var str = "[ <br />"
        for(var i = 0; i<array.length; i++){                //遍历数组格式的 JSON 对象
            str += "    {"
            for(var name in array[i]){                      //遍历数组元素对象
                str += name + " : " + array[i][name] + " , " //组合为 JSON 格式字符串
            }
            str = str.substring(0, str.length-1);           //清除掉最后一个字符
            str += "}, <br />";
        }
        str = str.substring(0, str.length-7);               //清除掉最后 7 个字符
        str += "<br />]";
        $("p").html(str);                                  //显示返回的 JSON 结构字符串
        return false;
    });
})
</script>
```

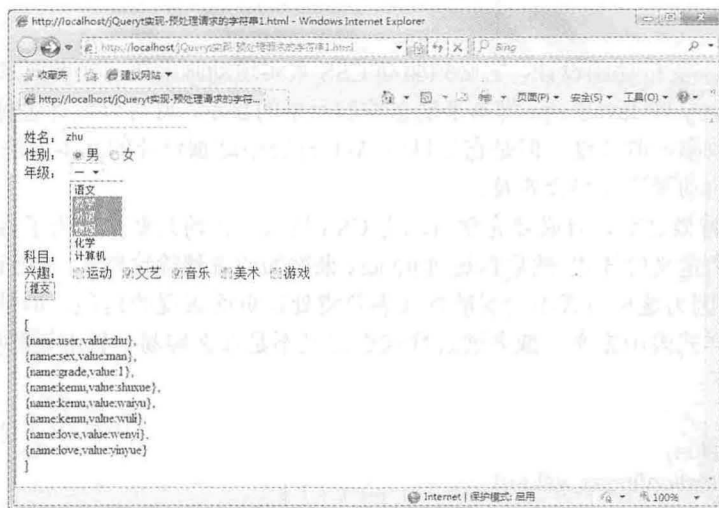



图 6.9 把请求的值转换为 JSON 对象结构

第 7 章

动画设计

( 视频讲解：1 小时 23 分钟)

对于 Web 设计来说，动画形式主要包括位置变化、形状变化和显示变化 3 种。位置变化主要通过元素的坐标值来控制。对于网页元素来说，形状变化主要是大小变化，这种形式主要依靠宽和高进行控制。而显示变化主要通过显示和隐藏属性进行控制，或者通过透明度进行控制。

在 JavaScript 开发中，动画效果会让操作更胜一筹。通过 jQuery，用户不仅能够轻松地为用户操作添加简单的视觉效果，甚至能创建更精致的动画。jQuery 效果确实能增添艺术性，一个元素逐渐滑入视野而不是突然出现时，带给人的美感是不言而喻的。此外，当页面发生变化时，通过效果吸引用户的注意力，则会显著增强页面的可用性。在本章中，将讲解 jQuery 动画设计效果，并将这些效果以有趣的方式组合起来。

7.1 CSS 动画设计基础

JavaScript 语言本身不支持动画设计，它必须驱动 CSS 来实现动画效果。在前面章节中也介绍了如何修改文档的外观。通过 jQuery 或 JavaScript 脚本来动态控制元素的显示，就可以设计各种复杂的动画效果。虽然这种动画效果的执行效率不敢恭维，但是它是目前 Web 开发中动画设计的基本方法。随着 HTML 5 语言的普及，相信更加高效的动画设计将会普及。

在接触 jQuery 各种特效之前，有必要先学习动态 CSS 技术。在前几章中，为了修改文档的外观，都是先在单独的样式表中为类定义好样式，然后再通过 jQuery 来添加或者移除这些类。一般而言，这都是为 HTML 应用 CSS 的首选方式，因为这种方式不会影响样式表负责处理页面表现的角色。但是，在有些情况下，要使用的样式可能没有在样式表中定义，或者通过样式表定义不是那么容易。针对这种情况，jQuery 提供了 `css()` 方法。具体用法如下：

```
css(propertyName)
css(propertyName, value)
css(propertyName, function(index, value))
css(map)
```

- ☑ 参数 `propertyName` 表示一个 CSS 属性名，以字符串形式表示，当 `css()` 方法只包含该参数时，表示为匹配的元素集合中获取第 1 个元素的样式属性值。参数 `value` 表示一个 CSS 属性名的值。
- ☑ 参数 `function(index,value)` 是一个返回设置值的函数，参数函数可以接收元素的索引位置和元素旧的样式属性值作为参数。
- ☑ 参数 `map` 表示一个名值对的对象直接量，结构类似于 `{name:value,name1:value1...}`。利用名值对对

象可以为匹配的元素设置一个或多个 CSS 属性。

【示例 1】 使用 jQuery 匹配文档中的段落文本元素 p，然后为它绑定 hover() 方法，定义鼠标移过时的动态样式，在该方法中包含两个回调函数，分别在鼠标移过和移出时调用。在这两个回调函数中使用 css() 定义段落文本的动态样式。演示效果如图 7.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").hover(
        function () {
            $(this).css({'background-color': 'yellow', 'color': 'red'});
        },
        function () {
            $(this).css({
                'background-color': '#fff',
                'color': 'rgb(0,0,255)'
            });
        }
    );
})
</script>
<title>上机练习</title>
</head>
<body>
<p>jQuery 是继 Prototype 之后又一个优秀的 JavaScript 框架。</p>
<p>它是轻量级的 JS 库，它兼容 CSS 3，还兼容各种浏览器（IE 6.0+，FF 1.5+，Safari 2.0+，Opera 9.0+）。</p>
</body>
</html>
```

【分解】

这个方法集获取方法（getter）和设置方法（setter）于一体。为取得某个样式属性的值，可以为这个方法传递一个字符串形式的属性名，如 css('backgroundColor')。对于由多个单词构成的属性名，jQuery 既可以解释连字符版的 CSS 表示法，如 background-color；也可以解释驼峰大小写形式的 DOM 表示法，如 backgroundColor。在设置样式属性时，css() 方法能够接收的参数有两种，一种是为它传递一个单独的样式属性和值，另一种是为它传递一个由名值对（属性—值对）构成的映射（map），用户可以将这些 jQuery 映射看成是 JavaScript 对象字面量。例如：

```
css('property','value')
css({property1:'value1','property-2':'value2'})
```

注意，如果属性值是数字值，不需要加引号，而字符串值需要加引号。但是，当使用映射表示法时，如果属性名使用驼峰大小写形式的 DOM 表示法，则可以省略引号。

【示例 2】 为 p 元素绑定 click 事件，当单击鼠标时，将会为当前元素调用 css() 方法，在方法中通过对原来值进行递增的形式，实现不断单击时递增文字大小。演示效果如图 7.2 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
```

```

<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function() {
        $(this).css({
            "font-size": function(index, value) {
                return parseFloat(value) * 1.2;
            }
        });
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<p>jQuery 是继 Prototype 之后又一个优秀的 JavaScript 框架。</p>
<p>它是轻量级的 JS 库，它兼容 CSS 3，还兼容各种浏览器（IE 6.0+，FF 1.5+，Safari 2.0+，Opera 9.0+）。</p>
</body>
</html>

```

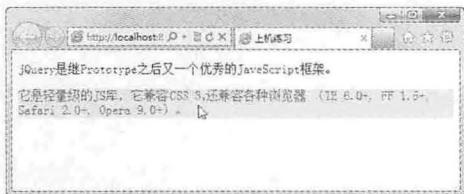


图 7.1 css()方法的应用 (1)

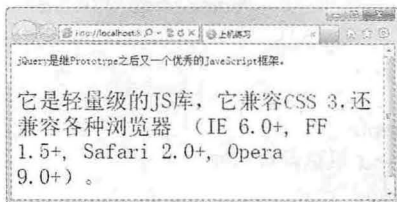


图 7.2 css()方法的应用 (2)

【分解】

使用 `css()` 的方式与 `addClass()` 的方式相同，将它连缀到选择符后面并绑定到某个事件。但是，假设希望每单击一次，文本的字体大小就会持续地递增或者递减。虽然为每次单击定义一个单独的类，然后迭代这些类也是可能的，但更简单明了的方法是每次都以前字体大小乘以一个设定的值，得到新的字体大小。

通过 `css('fontSize')` 可以轻而易举地取得当前的字体大小。不过，由于返回的值中既包含数字值也包含度量单位，所以需要把这两部分保存到各自的变量中，在乘出新的字体大小后，再重新加上单位。不过本示例直接通过 `css()` 方法的回调函数，就可以轻松解决这个繁琐的操作。

7.2 显隐动画

最简单的动画效果也许就是元素的显示和隐藏了。在 jQuery 中，使用 `show()` 方法可以显示元素，使用 `hide()` 方法可以隐藏元素。如果把 `show()` 和 `hide()` 方法配合起来，就可以设计最基本的显隐动画。

`show()` 方法的具体用法如下：

```

show()
show(duration, [callback])
show([duration], [easing], [callback])

```

- ☑ 参数 `duration` 表示一个字符串或者数字，决定动画将运行多久。
- ☑ 参数 `callback` 表示在动画完成时执行的函数。
- ☑ 参数 `easing` 表示一个用来表示使用哪个缓冲函数来过渡的字符串。

7.2.1 简单的显示和隐藏

基本的 `hide()` 和 `show()` 方法不带任何参数，可以把它们想象成类似 `css('display', 'string')` 方法的简写方式，其中 `string` 是适当的显示值。这两个方法的作用就是立即隐藏或显示匹配的元素集合，不带任何动画效果。

其中，`hide()` 方法会将匹配的元素集合的内联 `style` 属性设置为 `display:none`。但它的聪明之处是，它能够在把 `display` 的值变成 `none` 之前，记住原先的 `display` 值，通常是 `block` 或 `inline`。恰好相反，`show()` 方法会将匹配的元素集合的 `display` 属性，恢复为应用 `display:none` 之前的可见属性。

`show()` 和 `hide()` 的这种特性，使得它们非常适合隐藏那些默认的 `display` 属性在样式表中被修改的元素。例如，在默认情况下，`li` 元素具有 `display:block` 属性。但是，为了构建水平的导航菜单，它们可能会被修改成 `display:inline`。而在类似这样的 `li` 元素上面使用 `show()` 方法，不会简单地把它重置为默认的 `display:block`，因为那样会导致把 `li` 元素放到单独的一行中；相反，`show()` 方法会把它恢复为先前的 `display:inline` 状态，从而维持水平的菜单设计。

【示例 3】 使用 `for` 循环语句动态添加了 6 个 `div` 元素，并在内部样式表中定义盒子的尺寸、背景色、浮动显示，实现并列显示。然后为所有 `div` 元素绑定 `click` 事件，设计当单击 `div` 元素时，调用 `hide()` 方法隐藏该元素。演示效果如图 7.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    for (var i = 0; i < 5; i++) {
        $("<div>").appendTo(document.body);
    }
    $("div").click(function () {
        $(this).hide();
    });
})
</script>
<style>
div { background:red; width:100px; height:100px; margin:2px; float:left; }
</style>
<title>上机练习</title>
</head>
<body>
<div></div>
</body>
</html>
```

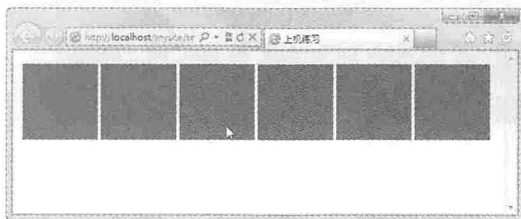


图 7.3 `hide()` 方法的应用

7.2.2 控制显示速度

当在 `show()` 或 `hide()` 中指定一个速度参数时, 就会产生动画效果, 即效果会在一个特定的时间段内发生。例如, `hide('speed')` 方法会同时减少元素的高度、宽度和不透明度, 直至这 3 个属性的值都达到 0, 与此同时会为该元素应用 CSS 规则 `display:none`。而 `show('speed')` 方法则会从上到下增大元素的高度, 从左到右增大元素的宽度, 同时从 0 到 1 增加元素的不透明度, 直至其内容完全可见。

对于 jQuery 提供的任何效果, 都可以指定 3 种速度参数: `slow`、`normal` 和 `fast`。使用 `show('slow')` 会在 0.6 秒内完成效果, `show('normal')` 是 0.4 秒, 而 `show('fast')` 则是 0.2 秒。要指定更精确的速度, 可以使用毫秒数值, 如 `show(850)`。注意, 与字符串表示的速度参数名称不同, 数值不需要使用引号。

【示例 4】 使用 `for` 循环语句动态添加了 6 个 `div` 元素, 并在内部样式表中定义盒子的尺寸、背景色、浮动显示, 实现并列显示。然后为所有 `div` 元素绑定 `click` 事件, 设计当单击 `div` 元素时, 调用 `hide()` 方法隐藏该元素。在 `hide()` 方法中设置隐藏显示的速度, 并定义在隐藏该 `div` 元素之后, 把当前元素移出文档。演示效果如图 7.4 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    for (var i = 0; i < 5; i++) {
        $("<div>").appendTo(document.body);
    }
    $("div").click(function () {
        $(this).hide(2000, function () {
            $(this).remove();
        });
    });
});
</script>
<style>
div { background:red; width:100px; height:100px; margin:2px; float:left; }
</style>
<title>上机练习</title>
</head>
<body>
<div></div>
</body>
</html>
```

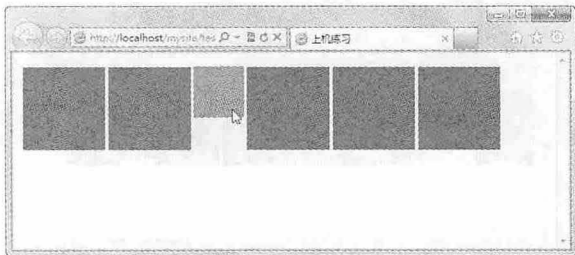


图 7.4 `hide()` 方法的应用

7.2.3 显隐切换

通过上面示例的演示可以看到，显示和隐藏是一对密不可分的动画形式。隐藏元素之后，一般都需要显示元素，反之亦然。在如此频繁的操作中，需要把 `show()` 方法和 `hide()` 方法与 `if` 条件结构结合起来，并定义一个显隐指示变量，显得很麻烦。jQuery 把这种繁琐的操作进行封装，定义了 `toggle()` 方法，下面就来详细进行讲解。

jQuery 定义的 `toggle()` 方法能够切换元素的可见状态。如果元素是可见的，将会把它切换为隐藏状态；如果元素是隐藏的，则把它切换为可见状态。具体用法如下：

```
toggle([duration], [callback])
toggle([duration], [easing], [callback])
toggle(showOrHide )
```

- ☑ 参数 `duration` 表示一个字符串或者数字，决定动画将运行多久。
- ☑ 参数 `callback` 表示在动画完成时执行的函数。
- ☑ 参数 `easing` 表示一个用来表示使用哪个缓冲函数来过渡的字符串。
- ☑ 参数 `showOrHide` 是一个布尔值指示是否显示或隐藏的元素。

如果没有参数，`toggle()` 方法是切换一个元素可见性的最简单的方法：

```
$('.target').toggle();
```

通过改变 CSS 的 `display` 属性，匹配的元素将被立即显示或隐藏，没有动画。如果元素是最初显示，它会被隐藏；如果是隐藏的，它会显示出来。`display` 属性将被存储并且在需要的时候可以恢复。如果一个元素的 `display` 值为 `inline`，然后是隐藏和显示，这个元素将再次显示 `inline`。

当提供一个持续时间参数时，`toggle()` 成为一个动画方法。`toggle()` 方法将匹配元素的宽度、高度以及不透明度，同时进行动画。当一个隐藏动画后，高度值达到 0 时，`display` 样式属性被设置为 `none`，以确保该元素不再影响页面布局。

`toggle()` 方法的持续时间是以毫秒为单位的，数值越大，动画越慢，不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。

如果提供回调函数参数，回调函数会在动画完成时调用，这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数，但是 `this` 是存在动画的 DOM 元素，如果多个元素一起做动画效果，值得注意的是，每执行一次回调函数，`this` 就匹配当前元素，而不是作为一个整体的动画一次。

【示例 5】 使用 `toggle()` 方法设计段落文本中的图像切换显示，同时添加了显示速度控制，以便更真实地显示动画显示过程。演示效果如图 7.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$( function(){
    $("button").click(function () {
        $("p").toggle("slow");
    });
});
</script>
<style type="text/css">
img { height:400px; }
</style>
<title>上机练习</title>
```



```

</head>
<body>
<p></p>
<button>显示和隐藏</button>
</body>
</html>

```

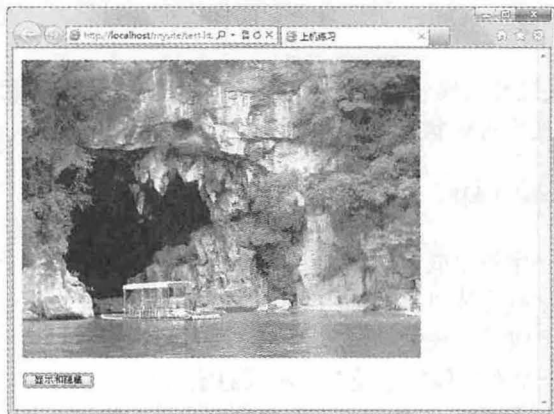


图 7.5 toggle()方法的应用

toggle()方法还可以接收多个参数。如果传入 true 或者 false 参数值,则可以设置元素显示或者隐藏,功能类似于 show()和 hide()方法。如果参数值为 true,则功能类似于调用 show()方法来显示匹配的元素;如果参数值为 false,则类似于调用 hide()方法来隐藏元素。

如果传入一个数值或者一个预定义的字符串,如 slow、normal 或者 fast,则表示在显隐切换时,以指定的速度动态显示匹配的显隐过程。

除了指定动画显隐的速度外,还可以在第 2 个参数指定一个回调函数,以备在动画演示完毕之后,调用该函数,以完成额外的任务。

7.2.4 折叠动画

折叠是网页设计中经常用到的效果,实现起来也很简单。但是为了技术的规范性和适应性,本节将对 JavaScript 代码进行简单的封装,实现在相同的结构和类样式下,都可以获得相同的折叠效果。演示效果如图 7.6 所示。



图 7.6 折叠效果

首先定制折叠面板的 HTML 结构, 这里选用 dl、dt 和 dd 3 个元素配合使用, 既符合语义性, 也方便管理。设想只要文档中包含 collapse 类样式的 dl 元素, 并确保只包含一个 dt 和 dd 子元素, 都可以拥有相同的折叠效果, 代码如下:

```
<dl class="collapse">
  <dt>标题栏</dt>
  <dd>内容框<br /></dd>
</dl>
```

以上面 HTML 结构为基础, 尝试使用 jQuery 来实现折叠效果。由于 jQuery 已经封装了 getElementsByClassName()、show() 和 hide() 方法, 所以可以直接调用。实现折叠效果的详细代码如下:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    //页面初始化处理函数
    var t = []; //定义空数组
    var dt = $("dl.collapse dt"); //获取类名为 collapse 的 dl 元素包含的所有 dt 子元素
    var dd = $("dl.collapse dd"); //获取类名为 collapse 的 dl 元素包含的所有 dd 子元素
    dt.each(function(i){ //遍历所有的 dt 元素, 并向函数传递遍历序号
        t[i] = false; //设置折叠初始状态
        $(dt[i]).click((function(i,dd){ //为当前 dt 元素绑定 click 事件处理函数
            return function(){ //返回一个闭包函数, 闭包能够存储传递进来的动态参数值
                if( t[i]){
                    $(dd).show(); //显示元素
                    t[i] = false;
                }else{
                    $(dd).hide(); //隐藏元素
                    t[i] = true;
                }
            }
        })(i,dd[i])); //向当前执行函数中传递参数
    })
})
</script>
<style type="text/css">
.collapse { border:solid 1px #ccc; margin:2px; float:left; }
.collapse dt { padding:8px 8px; background:#7FECAD url(images/green.gif) repeat-x; font-size:13px;
font-weight:bold; color:#71790C; border-bottom:solid 1px #efefef; cursor:pointer; }
.collapse dd { margin:0; padding:6px; }
.w1 {width:310px;}
.w2 {width:221px;}
.w3 {width:665px;}
</style>
<title>上机练习</title>
</head>
<body>
<dl class="collapse w1">
  <dt>音乐标签</dt>
  <dd></dd>
```

```

</dl>
<dl class="collapse w2">
  <dt>新歌 TOP100</dt>
  <dd></dd>
</dl>
<dl class="collapse w3">
  <dt>音乐掌门人</dt>
  <dd></dd>
</dl>
</body>
</html>

```

在 JavaScript 内建的核心中，Document 对象及 Element 对象只能有 3 种方式来获取元素：

```

getElementById('id')
getElementsByName('name')
getElementsTagName('tag')

```

这些方法分别根据元素的 id、name 和 tag 来获取元素。在 Web 文档中，id 具有唯一性，所以 getElementById(id)方法获取的对象也是唯一的，可以直接使用；而其他两个方法则会返回一个根据具有该属性的元素在文档中出现顺序排列的数组，使用时必须指定数组编号，如 array[0]表示第 1 个元素。

使用 jQuery 的设计思路与 JavaScript 设计思路完全相同，不过 jQuery 已经封装了 getElementsByName()、show()和 hide()方法，所以就会节省很多代码。同时，jQuery 使用 each()方法封装了 for 循环结构，实现快捷遍历文档节点。each()方法包含一个默认的参数，该参数可以传递遍历过程中元素的序列位置，以方便动态跟踪每个元素。

考虑到元素遍历的过程中，动态定位元素比较困难，这里使用了闭包函数存储元素的序列位置。由于在闭包中无法访问闭包函数外的对象，故还需要向其传递当前要操作的元素对象。

注意，在多层嵌套结构中大括号和小括号的使用，避免缺少使用小括号运算符，如 \$(dt[i]).click((function(i,dd){ //……})(i,dd[i]))。

7.2.5 树形动画

文件系统通常以层次结构列表形式显示，在层次结构列表里文件夹包含的内容相互嵌套，以便表示各种复杂的包含关系，这种类似树形动画的设计效果，在网页中经常见到，如目录导航，效果如图 7.7 所示。另外，在网页中看到的多级菜单也是一种经典的树形动画。

树形动画的第一步，就是要设计一个好的 HTML 结构。从语义性角度考虑，选择列表结构是最恰当的，当然也可以使用 div 和 span 元素实现相同的显示效果。列表结构在多层嵌套时，会自动显示出多层结构的关系，即使不使用 CSS 进行样式设计，也仍然让人一目了然。本案例的树形动画的 HTML 代码如下：

```

<ul class="tree">
  <li>首页</li>
  <li>新闻
    <ul>
      <li>国内新闻</li>
      <li>国际新闻</li>
    </ul>
  </li>
  <li>科技
    <ul>
      <li>桌面科技
      <li>移动科技
        <ul>
          <li>iPhone
          <li>HTC
          <li>Android
        </ul>
      <li>应用科技
    </ul>
  </li>
  <li>社会

```

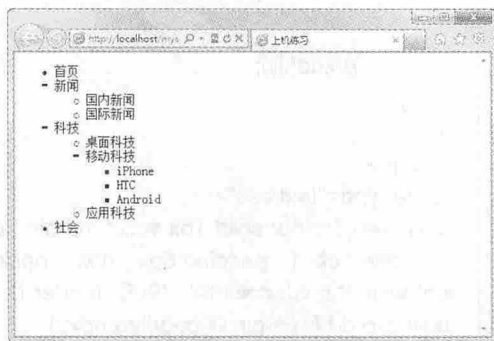


图 7.7 树形动画


```

        </ul>
    </li>
    <li>科技
        <ul>
            <li>桌面科技</li>
            <li>移动科技
                <ul>
                    <li>iPhone</li>
                    <li>HTC </li>
                    <li>Android</li>
                </ul>
            </li>
            <li>应用科技</li>
        </ul>
    </li>
    <li>社会</li>
</ul>

```

整个树形动画包含在 ul 容器中，每一个 li 元素作为一个选项进行呈现，不同层次的结构分别以 ul 子元素进行包裹，从而实现层层嵌套的关系。

使用 JavaScript 直接设计树形动画的思路如下：

先获取树形动画中的所有 li 元素，因为 li 元素代表一个选项，不管该选项处于什么层次位置。然后，遍历 li 元素集合，在遍历过程中检测当前 li 元素是否包含 ul 元素。如果包含 ul 元素，则设置临时标识变量 b 为 true，否则设置变量 b 为 false。

☑ 如果 b 为 true，则设置当前 li 元素的样式（如鼠标样式、列表项目符号），并获取 li 元素包含的第 1 个 ul 元素，并隐藏该 ul 元素。然后为当前 li 元素绑定 click 事件处理函数。在该事件处理函数中，根据 ul 元素是否显示为条件，分别隐藏或者显示 ul 元素，同时动态修改当前 li 元素的样式。

☑ 如果 b 为 false，则设置当前 li 元素的样式为默认状态。

为了防止单击当前 li 元素的子元素时，也可能会触发 click 事件，应该检测当前单击的元素是否为 li 元素。为此，可以使用 Event 对象的 target（兼容非 IE 浏览器）或 srcElement（兼容 IE 浏览器）属性进行判断。完整的 JavaScript 脚本代码如下：

```

<script type="text/javascript">
window.onload = function(){
    var li = document.getElementsByTagName("li");           //页面初始化处理函数
    //获取页面中的所有 li 元素
    var t=[];                                                //定义临时数组
    //遍历数组
    for(var i = 0; i < li.length; i ++ ){
        var child = li[i].childNodes;                       //获取当前 li 元素包含的所有子节点
        var b = false;                                       //定义临时变量，并初始化为 false
        //遍历当前 li 元素包含的节点，并检测是否包含 ul 元素
        for(var j=0; j<child.length;j++){
            if(child[j].nodeType == 1 && child[j].nodeName.toLowerCase() == "ul")
                b = true;                                     //如果 li 元素包含 ul 元素，则设置 b 为 true
        }
        if(b){
            li[i].style.cursor = 'pointer';                 //如果 li 元素包含 ul 元素
            //定义当前 li 元素的鼠标指针样式为手形
            li[i].style.listStyleImage = 'url(images/+.gif)'; //修改当前 li 元素的选项列表图标形状
            var ul = li[i].getElementsByTagName("ul")[0];    //获取第 1 个 ul 子元素
            ul.style.display = "none";                       //隐藏第 1 个 ul 元素
            t[i] = true;                                       //设置当前序号位置的数组元素的值为 true
            li[i].onclick = (function(o,li,i){              //绑定 click 单击事件处理函数
                return function(e){                          //返回闭包函数

```



```

        if(li == e.target || li == window.event.srcElement){ //如果当前元素就是事件触发的目标对象，则允许执行。这样做的目的是防止单击当前 li 元素的子元素时，也触发 click 事件
            if( t[i]){ //如果当前数组元素值为 true
                o.style.display = ""; //恢复显示 ul 元素
                li.style.listStyleImage = 'url(images/-.gif)'; //修改 li 元素项目列表符号
                t[i] = false; //切换当前数组元素值为 false
            }
            else{ //如果当前数组元素值为 false
                o.style.display = "none"; //隐藏显示 ul 元素
                li.style.listStyleImage = 'url(images/+.gif)'; //修改 li 元素项目列表符号
                t[i] = true; //切换当前数组元素值为 true
            }
        }
        if ( e && e.stopPropagation ) //兼容非 IE 浏览器
            e.stopPropagation(); //阻止事件传播
        else //兼容 IE 浏览器
            window.event.cancelBubble = true; //阻止事件传播
        return false; //避免触发默认事件
    }
})(ul,li[i],i); //调用当前函数，并传递当前 li 元素及其包含的第 1 个 ul 元素，以及当前 li 元素位置的序号
}
else{ //如果 li 元素不包含 ul 元素
    li[i].style.cursor = 'default'; //恢复 li 元素的鼠标默认样式
    li[i].style.listStyleImage = 'none'; //恢复 li 元素的默认列表项目符号
}
}
}
</script>

```

根据 JavaScript 设计思路，下面尝试使用 jQuery 来实现相同的设计效果。详细代码如下：

```

<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$( function(){ //页面初始化处理函数
    $('li:has(ul)') .click( function( event ){ //如果 li 元素包含 ul 元素，则绑定 click 事件
        if ( this == event.target ) { //如果当前 li 元素就是事件触发的目标对象
            if ( $( this ).children().is( ':hidden' ) ) { //如果当前 li 元素的子元素隐藏，则修改 li 元素的项目列表符号，并显示所有子元素
                $( this ).css( 'list-style-image', 'url(images/-.gif)' ).children().show();
            }
            else { //否则修改 li 元素的项目列表符号，并隐藏所有子元素
                $( this ).css( 'list-style-image', 'url(images/+.gif)' ).children().hide();
            }
        }
        return false;
    }) .css( { //设置包含 ul 子元素的 li 元素的样式
        cursor : 'pointer', //设置鼠标样式为手形
        'list-style-image' : 'url(images/+.gif)' //设置项目列表符号为减号样式
    }) .children().hide(); //隐藏当前 li 元素的所有子元素
    $('li:not(:has(ul))') .css( { //如果 li 元素没有包含 ul 元素，则设置如下样式
        cursor : 'default', //恢复默认的鼠标样式
        'list-style-image' : 'none' //恢复默认的项目列表符号
    });
});
</script>

```

7.2.6 选项卡动画

如果说树形动画体现的是一种多层次结构关系,那么选项卡体现的就是索引结构关系。通过 Tab 索引可以快速定位到相应的模块选项,选项卡在分类组织方面功能显著。在 Web 开发中,以选项卡形式设计的页面或者模块比较常见,如图 7.8 所示。

选项卡的结构通常按二叉型进行设计,框 1 (分支一) 负责组织 Tab 标题内容,而框 2 (分支二) 负责组织每个选项卡对应的显示内容。在设计时,为了方便控制,应确保 Tab 标题序列与内容序列一一对应,这样可以方便程序进行控制。本案例的 HTML 结构如下:

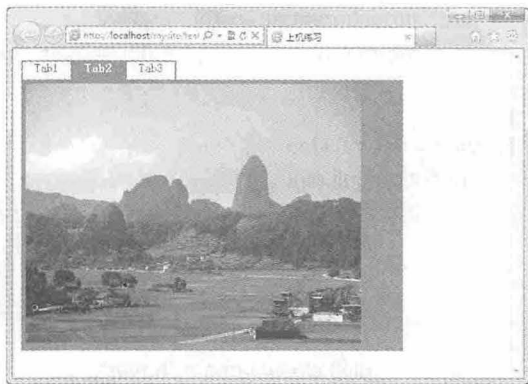


图 7.8 选项卡

```
<div class="tab">
  <ul> <!-- 选项卡标题框 -->
    <li>Tab1</li>
    <li>Tab2</li>
    <li>Tab3</li>
  </ul>
  <ol> <!-- 选项卡内容框 -->
    <li></li>
    <li></li>
    <li></li>
  </ol>
</div>
```

使用 JavaScript 直接设计选项卡的思路是:先使用 CSS 设计 4 对类样式,分别用来控制标题栏和内容框的显隐样式。使用 JavaScript 设计在默认状态下标题栏和内容框的类样式,然后通过遍历方式为每个标题栏绑定 mouseover 事件处理函数,设计当鼠标经过标题栏时,隐藏所有内容框,修改所有标题的类样式,并显示该标题栏的样式和现实所对应的内容框。使用 JavaScript 实现选项卡功能的完整代码如下:

```
<script type="text/javascript">
//省略了为 Document 对象扩展的 getElementsByClassName ()方法
//详细讲解请参阅第 7.1.3 节的内容
window.onload = function(){
  var tab = document.getElementsByClassName("tab")[0]; //获取选项卡的外框
  var ul = tab.getElementsByTagName("ul")[0]; //获取选项卡标题栏的外框
  var ol = tab.getElementsByTagName("ol")[0]; //获取选项卡内容框的外框
  var uli = ul.getElementsByTagName("li"); //获取所有标题栏选项
  var oli = ol.getElementsByTagName("li"); //获取所有内容选项
  for(var i=0; i<uli.length; i++){ //遍历标题栏选项
    uli[i].className = "normal"; //设置所有标题栏选项的类样式为普通样式
  }
  for(var i=0; i<oli.length; i++){ //遍历内容框选项
    oli[i].className = "none"; //设置所有内容框选项的类样式为隐藏
  }
  uli[0].className = "hover"; //设置第 1 个标题栏选项为凸起显示
  oli[0].className = "show"; //设置第 1 个内容框选项为显示出来
}
```



```

var addEvent=function(e, fn) { //自定义绑定 mouseover 事件函数
    if(document.addEventListener){ //兼容非 IE 浏览器
        return e.addEventListener("mouseover", fn, false);
    }
    else if(document.attachEvent){ //兼容 IE 浏览器
        return e.attachEvent("onmouseover", fn);
    }
};
for(var j = 0; j < uli.length; j ++){ //遍历标题栏选项
    (function(j,uli,oli){ //调用匿名函数
        addEvent(uli[j], function(){ //为当前标题栏选项元素绑定 mouseover 事件
            for(var n = 0; n < oli.length; n ++){ //遍历内容框选项
                uli[n].className = "normal"; //恢复所有标题栏选项为普通显示状态
                oli[n].className = "none"; //隐藏所有内容框选项
            }
            uli[j].className = "hover"; //设置当前标题栏为凸起效果
            oli[j].className = "show"; //显示当前标题栏对应的内容框选项
        });
    })(j,uli,oli); //把当前序号、标题栏选项数组和内容框选项数组传递进去
}
</script>

```

根据 JavaScript 设计思路, 使用 jQuery 实现相同的设计效果, 编写的代码会非常简洁, 具体代码如下:

```

<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$( function(){ //页面初始化事件处理函数
    var $uli = $(".tab ul li"); //获取所有标题栏选项元素
    var $oli = $(".tab ol li"); //获取所有内容框选项元素
    $uli.addClass("normal"); //为所有标题栏选项元素添加普通类样式
    $oli.addClass("none"); //为所有内容框选项元素添加隐藏类样式
    $uli[0].className = "hover"; //初始化第 1 个标题栏选项显示为凸起效果
    $oli[0].className = "show"; //初始化第 1 个内容框选项显示出来
    $uli.each(function(n){ //遍历所有标题栏选项
        $(this).mouseover(function(){ //为每个选项绑定 mouseover 事件处理函数
            $uli.removeClass().addClass("normal"); //移出所有标题栏选项的类样式, 恢复普通显示状态
            $(this).removeClass().addClass("hover");//移出所有类样式, 为当前标题栏选项元素设置凸起显示状态
            $oli.removeClass().addClass("none"); //移出所有内容框选项的类样式, 恢复隐藏显示状态
            $($oli[n]).removeClass().addClass("show"); //移出所有类样式, 为当前内容框选项元素设置显示状态
        })
    })
});
</script>

```

7.3 滑 动 动 画

滑动效果有两种: 匀速滑动和变速滑动。设计匀速运动效果只需要使用 JavaScript 动态控制均匀移动元素的位置即可, 而变速运动就需要掌握一些简单的算法。但是不管是哪种滑动效果, 使用 JavaScript 实现滑动效果一般是根据如下设计思路进行设计:

(1) 先获取元素的 `display` 样式属性, 判断元素是否为隐藏状态, 否则返回, 不允许执行滑动显示。

(2) 借助 `css()` 方法设置元素 `display` 样式属性为默认值 (即非隐藏), `visibility` 属性值为 `hidden` (即隐藏显示)。这时就可以利用 `offset()` 获取元素的坐标值以及元素的宽和高的尺寸值。

(3) 复制当前元素, 并清除该元素包含的子节点, 把这个复制的元素作为盒子, 把当前元素移动到该盒子中, 并删除原元素。

(4) 设置盒子元素的 `overflow` 样式属性为 `hidden`, `display` 为显示状态, 宽度等于元素的宽度, 高度为 0, 这样就可以通过逐步增大盒子元素的高度, 逐步显示盒子包含的元素, 从而设计出元素逐步滑动显示的效果。

(5) 当元素完全显示之后, 则清除循环执行命令, 并把这个临时的盒子元素删除掉, 同时移出该盒子包含的当前元素, 把它移到盒子相邻的位置, 这样删除了临时盒子元素之后, 又恢复了文档最初的结构。

最后, 调用回调函数, 整个动画演示过程结束。

jQuery 提供了简单的滑动方法以及自定义动画函数, 借助 jQuery 的这些功能, 我们能够完成各种网页动画的设计要求。

7.3.1 显隐滑动效果

`slideDown()` 和 `slideUp()` 是 jQuery 定义的两个滑动方法, 它们分别是向下滑动和向上滑动, 相当于缓慢舒展和缓慢收缩元素对象。如果灵活配合 `slideDown()` 和 `slideUp()` 方法, 可以设计很多奇妙、动感的滑动效果。这两个方法的具体用法如下:

```
slideDown([duration], [callback])
slideDown([duration], [easing], [callback])
slideUp([duration], [callback])
slideUp([duration], [easing], [callback])
```

☑ 参数 `duration` 为一个字符串或者数字, 用来定义动画将运行多久。

☑ 参数 `easing` 为一个用来表示使用哪个缓冲函数来过渡的字符串。

☑ 参数 `callback` 表示在动画完成时执行的函数。

`slideDown()` 和 `slideUp()` 方法将给匹配元素的定义滑动显示或者滑动隐藏动画效果, 它主要通过改变高度的值来实现。其中 `slideDown()` 能够导致页面的下面部分滑下去, 弥补了显示物品的方式。而 `slideUp()` 方法导致页面的下面部分滑上去, 弥补了显示物品的方式。一旦高度达到 0, `display` 样式属性将被设置为 `none`, 以确保该元素不再影响页面布局。

`slideDown()` 和 `slideUp()` 方法的持续时间是以毫秒为单位的, 数值越大, 动画越慢, 不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。如果提供任何其他字符串, 或者这个 `duration` 参数被省略, 那么默认使用 400 毫秒的延时。

如果提供回调函数参数, 回调函数会在动画完成时调用, 这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数, 但是 `this` 是存在动画的 DOM 元素, 如果多个元素一起做动画效果, 每执行一次回调匹配的元素, 而不是作为一个整体的动画一次。

【示例 6】 有 3 个按钮和 3 个文本框, 当单击按钮时将自动隐藏按钮后面的文本框, 且是以滑动方式逐渐隐藏, 隐藏之后会在底部 `<div id="msg">` 信息框中显示提示信息。演示效果如图 7.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
```



```

$("button").click(function () {
    $(this).parent().slideUp("slow", function () {
        $("#msg").text($("#button", this).text() + "已经实现。");
    });
});
});
</script>
<style type="text/css">
div { margin:2px; }
</style>
<title>上机练习</title>
</head>
<body>
<div>
    <button>隐藏文本框 1</button>
    <input type="text" value="文本框 1" />
</div>
<div>
    <button>隐藏文本框 2</button>
    <input type="text" value="文本框 2" />
</div>
<div>
    <button>隐藏文本框 3</button>
    <input type="text" value="文本框 3" />
</div>
<div id="msg"></div>
</body>
</html>

```

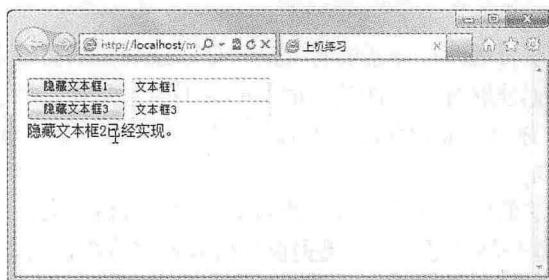


图 7.9 显隐滑动的应用

注意，`slideDown()`方法仅适用于被隐藏的元素。如果为已显示的元素调用 `slideDown()`方法，是看不到效果的。而 `slideUp()`方法正好相反，它可以把显示的元素缓慢地隐藏起来。`slideDown()`和 `slideUp()`方法正像卷帘，`slideDown()`方法能够缓慢地展开帘子，而 `slideUp()`方法能够缓慢地收缩帘子。通俗描述，`slideDown()`方法作用于隐藏元素，而 `slideUp()`方法作用于显示元素，两者功能和效果截然相反。

`slideDown()`和 `slideUp()`方法可以包含两个可选的参数，第 1 个参数用于设置滑动的速度，可以设置预定义字符串，如 `slow`、`normal` 和 `fast`，或者传递一个数值，表示动画时长的毫秒数。第 2 个可选参数表示一个回调函数，当动画完成之后，将调用该回调函数。

7.3.2 显隐切换滑动

与 `toggle()`方法的功能相似，jQuery 为滑动效果也设计了一个切换滑动的方法——`slideToggle()`。`slideToggle()`

方法的使用与 `slideDown()` 和 `slideUp()` 方法的使用相同,但是它综合了 `slideDown()` 和 `slideUp()` 方法的动画效果,可以在滑动中切换显示或隐藏元素。具体用法如下:

`slideToggle([duration], [callback])`

`slideToggle([duration], [easing], [callback])`

- ☑ 参数 `duration` 为一个字符串或者数字,决定动画将运行多久。
- ☑ 参数 `easing` 是一个用来表示使用哪个缓冲函数来过渡的字符串。
- ☑ 参数 `callback` 表示在动画完成时执行的函数。

`slideToggle()` 方法将为匹配元素实现高度变化的动画,这会导致页面布局容易发生变化。`display` 属性将被存储并且需要的时候可以恢复。如果一个元素的 `display` 值为 `inline`, 然后是隐藏和显示,这个元素将再次显示 `inline`。当一个隐藏动画后,高度值达到 0 时, `display` 样式属性被设置为 `none`, 以确保该元素不再影响页面布局。

`slideToggle()` 方法的持续时间是以毫秒为单位的,数值越大,动画越慢,不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。

如果提供回调函数参数,回调函数会在动画完成时调用。这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数,但在回调函数中, `this` 是隐性存在的,它代表当前动画的 DOM 元素,如果多个元素一起做动画效果,每执行一次回调匹配的元素,而不是作为一个整体的动画一次。

【示例 7】 在本示例中有一个按钮,当单击按钮时将自动隐藏部分 `div` 元素,同时显示被隐藏的 `div` 元素。演示效果如图 7.10 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#aa").click(function () {
        $("div:not(.still)").slideToggle("slow", function () {
            var n = parseInt($("#span").text(), 10);
            $("#span").text(n + 1);
        });
    });
});
</script>
<style type="text/css">
div { background:#b977d1; margin:3px; width:60px; height:60px; float:left; }
div.still { background:#345; width:5px; }
div.hider { display:none; }
span { color:red; }
p { clear: left; }
</style>
<title>上机练习</title>
</head>
<body>
<div></div>
<div class="still"></div>
<div style="display:none;"> </div>
<div class="still"></div>
```



```

</div></div>
<div class="still"></div>
<div class="hider"></div>
<div class="still"></div>
<div class="hider"></div>
<div class="still"></div>
<div></div>
<p>
  <button id="aa">滑动切换</button>
  共计滑动切换<span>0</span>个 div 元素。</p>
</body>
</html>

```

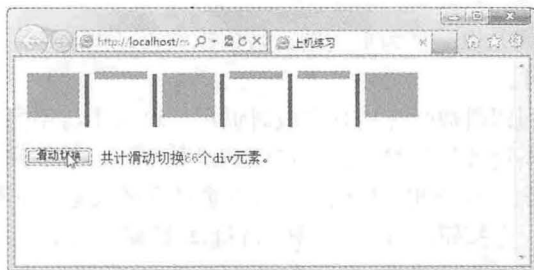


图 7.10 显隐切换滑动的应用

7.4 渐变效果

渐变效果是通过不透明度的变化来实现匹配元素的淡入和淡出动画。与滑动效果相比，渐变效果只调整元素的不透明度，也就是说元素的高度和宽度不会发生变化。

实现渐变效果的技术核心是各个浏览器支持的不透明度特效。但是由于不同浏览器支持的不透明度特效方法不同，导致在设计渐隐渐显时，首先应该考虑浏览器兼容性问题。IE 浏览器支持 `filters` 滤镜集，而非 IE 浏览器支持 `style.opacity` 属性。另外，IE 的 `opacity` 属性值范围为 0~100。其中，0 表示完全透明，100 表示不透明。而 `style.opacity` 属性的取值范围是 0~1。其中，0 表示完全透明，1 表示不透明。为了兼容不同浏览器，先为元素扩展一个方法，用来设置元素的不透明度。

jQuery 为元素渐隐和渐显定义了 3 个方法：`fadeIn()`、`fadeOut()` 和 `fadeTo()`。下面分别进行详细说明。

7.4.1 淡入和淡出

`fadeIn()` 和 `fadeOut()` 方法的具体用法如下：

```

fadeIn([duration], [callback])
fadeIn([duration], [easing], [callback])
fadeOut([duration], [callback])
fadeOut([duration], [easing], [callback])

```

- ☑ 参数 `duration` 为一个字符串或者数字，该参数决定动画将运行多久。
- ☑ 参数 `easing` 是一个用来表示使用哪个缓冲函数来过渡的字符串。
- ☑ 参数 `callback` 是一个在动画完成时执行的函数。

`fadeOut()` 方法通过匹配元素的透明度做动画效果。一旦透明度达到 0，`display` 样式属性将被设置为 `none`，以确保该元素不再影响页面布局。

fadeOut()和fadeIn()方法的延时时间是以毫秒为单位的,数值越大,动画越慢,不是越快。字符串 fast 和 slow 分别代表 200 和 600 毫秒的延时。如果提供任何其他字符串,或者这个 duration 参数被省略,那么默认使用 400 毫秒的延时。

如果提供回调函数参数,回调函数会在动画完成时调用。这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数,但是 this 是存在动画的 DOM 元素,如果多个元素一起做动画效果。

【示例 8】 为段落文本中的 span 元素绑定 hover 事件,设计鼠标移过时动态背景效果,同时绑定 click 事件,当单击 span 元素时,将渐隐该元素,并把该元素包含的文本传递给 div 元素,实现隐藏提示信息效果。演示效果如图 7.11 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$( function(){
    $("span").click(function () {
        $(this).fadeOut(1000, function () {
            $("div").text("“" + $(this).text() + "” 已经隐藏。");
            $(this).remove();
        });
    });
    $("span").hover(
        function () {
            $(this).addClass("hilite");
        },
        function () {
            $(this).removeClass("hilite");
        }
    );
});
</script>
<style type="text/css">
span { cursor:pointer; }
span.hilite { background:yellow; }
div { display:inline; color:red; }
</style>
<title>上机练习</title>
</head>
<body>
<h3>隐藏提示: <div></div></h3>
<p>雨, <span>轻薄浅落</span>, <span>丝丝缕缕</span>, <span>幽幽怨怨</span>。不知何时起,细腻的心
莫名地爱上了阴雨天。也许,雨天是思念的<span>风铃</span>,雨飘下,铃便响。伸出薄凉的手掌,雨轻弹地
滴落在掌心,<span>凉意</span>,遍布全身;<span>怀念</span>,张开翅膀;<span>眼角</span>,已感湿润。
</p>
</body>
</html>

```

fadeIn()方法能够实现所有匹配元素的淡入效果,并在动画完成后可选地触发一个回调函数。而 fadeOut()方法正好相反,它能够实现所有匹配元素的淡出效果。通过示例 8 可以看到,fadeIn()和 fadeOut()方法与 slideDown()和 slideUp()方法的用法是完全相同的,它们都可以包含两个可选参数。第 1 个参数表示动画持

续的时间，以毫秒为单位。另外，还可以使用预定义字符串 `slow`、`normal` 和 `fast`，使用这些特殊的字符串可以设置动画以慢速、正常速度和快速进行演示。第 2 个参数表示回调函数，该参数为可选参数，用来在动画演示完毕之后被调用。例如，在下面示例中，当单击按钮之后调用 `div` 元素的 `fadeIn()` 方法，逐步显示隐藏的元素，当显示完成之后，再次调用回调函数。

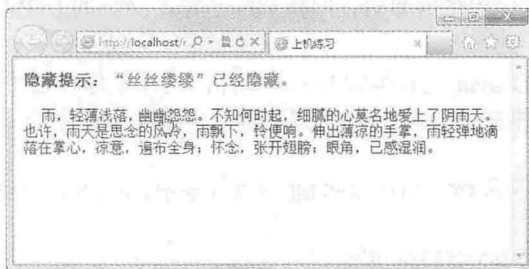


图 7.11 淡入和淡出应用

注意，与 `slideDown()` 和 `slideUp()` 方法的用法相同，`fadeIn()` 方法只能够作用于被隐藏的元素，而 `fadeOut()` 方法只能够作用于显示的元素。

7.4.2 设置淡出透明效果

`fadeTo()` 方法能够把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，并在动画完成后可选地触发一个回调函数。具体用法如下：

`fadeTo(duration, opacity, [callback])`

`fadeTo([duration], opacity, [easing], [callback])`

- ☑ 参数 `duration` 为一个字符串或者数字，决定动画将运行多久。
- ☑ 参数 `opacity` 是一个 0~1 之间的数字，表示目标透明度。
- ☑ 参数 `easing` 是一个用来表示使用哪个缓冲函数来过渡的字符串。
- ☑ 参数 `callback` 为在动画完成时执行的函数。

该方法的延时时间是以毫秒为单位的，数值越大，动画越慢，不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。如果提供任何其他字符串，或者这个 `duration` 参数被省略，那么默认使用 400 毫秒的延时。和其他效果方法不同，`.fadeTo()` 需要明确地指定 `duration` 参数。

如果提供回调函数参数，回调函数会在动画完成时调用。这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数，但是 `this` 是存在动画的 DOM 元素，如果多个元素一起做动画效果，值得注意的是，这个回调函数在每个匹配的元素上执行一次，不是这个动画作为一个整体。

【示例 9】把图像逐步调整到不透明度为 0.4 的显示效果，演示效果如图 7.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").fadeTo(2000,0.4);
    })
})
</script>
```



```

<title>上机练习</title>
</head>
<body>
<input type="button" value="渐隐渐显效果" />
<div></div>
</body>
</html>

```



图 7.12 设置淡出透明效果

注意, `fadeOut()` 方法仅能够作用于显示的元素, 对于被隐藏的元素来说是无效的。

7.4.3 渐变切换

与 `toggle()` 方法的功能相似, jQuery 为渐变效果也设计了一个渐变切换的方法——`fadeToggle()`。`fadeToggle()` 方法的用法与 `fadeIn()` 和 `fadeOut()` 方法的用法相同, 但是它综合了 `fadeIn()` 和 `fadeOut()` 方法的动画效果, 可以在渐变中切换显示或隐藏元素。具体用法如下:

`fadeToggle([duration], [callback])`

`fadeToggle([duration], [easing], [callback])`

- ☑ 参数 `duration` 为一个字符串或者数字, 决定动画将运行多久。
- ☑ 参数 `easing` 是一个用来表示使用哪个缓冲函数来过渡的字符串。
- ☑ 参数 `callback` 表示在动画完成时执行的函数。

`fadeToggle()` 方法的持续时间是以毫秒为单位的, 数值越大, 动画越慢, 不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。

如果提供回调函数参数, 回调函数会在动画完成时调用。这对于将不同的动画串联在一起按顺序排列是非常有用的。这个回调函数不设置任何参数, 但是 `this` 是存在动画的 DOM 元素, 如果多个元素一起做动画效果, 值得注意的是, 每执行一次回调匹配的元素, 而不是作为一个整体的动画一次。

【示例 10】 在本示例中显示两个按钮, 当单击这两个按钮时, 会切换渐变显示或者隐藏下面的图像, 在第 2 个按钮的 `click` 事件处理函数中调用 `fadeToggle()` 方法时, 传递一个回调函数, 在这个函数中将每次单击按钮 2 的信息追加到 `div` 元素中。演示效果如图 7.13 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button:first").click(function() {

```

```

    $("img:first").fadeToggle("slow", "linear");
  });
  $("button:last").click(function () {
    $("img:last").fadeToggle("fast", function () {
      $("#log").append("<div>单击按钮 2</div>");
    });
  });
})
</script>
<title>上机练习</title>
</head>
<body>
<button>控制按钮 1</button>
<button>控制按钮 2</button>
<p></p>
<div id="log"></div>
</body>
</html>

```



图 7.13 渐变切换效果

7.5 复杂动画

在绑定到 jQuery 核心的效果中,只有 show()和 hide()会同时修改多个样式属性:高度、宽度和不透明度,其他效果则只修改一种属性。

- ☑ fadeIn()和 fadeOut(): 不透明度。
- ☑ fadeTo(): 不透明度。
- ☑ slideDown()和 slideUp(): 高度。

jQuery 也提供了一个强大的 animate()方法,通过该方法可以创建包含多重效果的自定义动画,即执行一个 CSS 属性设置的自定义动画。具体用法如下:

```

animate(properties, [duration], [easing], [callback])
animate(properties, options)

```

- ☑ 参数 properties 表示一组 CSS 属性,动画将朝着这组属性移动。
- ☑ 参数 duration 表示一个字符串或者数字,决定动画将运行多久。
- ☑ 参数 easing 定义要使用的擦除效果的名称,但是需要插件支持,默认 jQuery 提供 linear 和 swing。
- ☑ 参数 callback 表示在动画完成时执行的函数。

- ☑ 参数 options 表示一组包含动画选项的值的集合。支持的选项如下。
 - duration: 3 种预定速度之一的字符串, 如 slow、normal 或者 fast, 或者表示动画时长的毫秒数值, 如 1000。默认值为 normal。
 - easing: 要使用的擦除效果的名称, 需要插件支持, jQuery 提供 linear 和 swing 两个值。默认值为 swing。
 - complete: 在动画完成时执行的函数。
 - step: 每步动画执行后调用的函数。
 - queue: 设定为 false, 将使此动画不进入动画队列。默认值为 true。
 - specialEasing: 一组一个或多个通过相应的参数和相对简单函数定义的 CSS 属性。

7.5.1 模拟 show()方法的效果

show()方法能够显示隐藏的元素, 它会同时修改元素的宽度、高度和不透明度属性。因此, 事实上它只是 animate()方法的一种内置了特定样式属性的简写形式。如果想通过 animate()得到同样的效果, 那么就非常简单。

【示例 11】 使用 hide()方法隐藏图像, 然后当单击按钮时, 将会触发 click 事件, 然后缓慢显示图像。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("img").hide();
    $("button").click(function () {
        $("img").show('slow');
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<button>控制按钮 1</button>
<p></p>
</body>
</html>
```

针对示例 11, 可以使用 animate()方法进行模拟, 具体代码如下。演示效果如图 7.14 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("img").hide();
    $("button").click(function () {
        $("img").animate({
```



```

        height:'show',
        width:'show',
        opacity:'show'
    },'show');
});
})
</script>
<title>上机练习</title>
</head>
<body>
<button>控制按钮 1</button>
<p></p>
</body>
</html>

```



图 7.14 show 效果

animate()方法拥有一些简写的参数值,这里使用简写的 show 将高度、宽度等恢复到了它们被隐藏之前的值。当然也可以使用 hide、toggle 或其他任意数字值。

7.5.2 自定义动画

animate()方法可以用于创建自定义动画,该方法的关键就在于指定动画的形式以及动画结果样式属性的对象。

【示例 12】 设计当单击按钮时,图像的大小被放大到原始大小。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").click(function(){
        $("img").animate({
            width: "100%",
            height: "100%"
        }, 1000 );
    })
})

```

```

</script>
<title>上机练习</title>
</head>
<body>
<button>控制按钮 1</button>
<p></p>
</body>
</html>

```

animate()方法包含 4 个参数,第 1 个参数是一组包含作为动画属性和终值的样式属性和其值的集合。形式类似下面的代码:

```

{
    width: "90%",
    height: "100%",
    fontSize: "10em",
    borderWidth: 10
}

```

这个集合对象中每个属性都表示一个可以变化的样式属性,如 height、top、opacity 等。注意,所有指定的属性必须采用骆驼命名形式,如 marginLeft,而不是 margin-left。这些属性的值表示这个样式属性到多少时动画结束。

如果属性值是一个数值,样式属性就会从当前的值渐变到指定的值。如果使用的是 hide、show 或 toggle 等特定字符串值,则会为该属性调用默认的动画形式。例如,在下面示例中,在一个动画中同时应用 4 种类型的效果,放大文本大小、扩大元素宽和高、同时多次单击,都可以在高度和不透明度之间来回切换显示 p 元素。当然,可以添加更多的动画样式,以设计复杂的动态效果。

```

<script type="text/javascript">
$(function(){
    $("button").click(function(){
        $("p").animate({
            width: "200%",
            height: "200%",
            fontSize: "5em",
            height: 'toggle',
            opacity: 'toggle'
        }, 1000 );
    })
})
</script>

```

第 2 个参数表示动画持续的时间,以毫秒为单位,也可以设置预定义字符串,如 slow、normal 和 fast。在 jQuery 1.3 中,如果第 2 个参数设置为 0,则表示直接完成动画,而在以前版本中则会执行默认动画。

第 3 个参数表示要使用的擦除效果的名称,这是一个可选参数,要使用该参数,则需要插件支持。默认 jQuery 提供 linear 和 swing 特效。

第 4 个参数表示回调函数,表示在动画演示完毕之后将要调用的函数。

【示例 13】 使 div 向左右平滑移动。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">

```

```

$(function(){
    $("input").eq(0).click(function(){
        $("div").animate({
            left: "-100px"
        }, 1000)
    })
    $("input").eq(1).click(function(){
        $("div").animate({
            left: "+200px"
        }, 1000)
    })
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="向左运动" /><input type="button" value="向右运动" />
<div style="position:absolute;left:200px; border:solid 1px red;">自定义动画</div>
</body>
</html>

```

注意，要想使 div 元素能够自由移动，必须设置它的定位方式为绝对定位、相对定位或者固定定位。如果是静态定位，则移动动画是无效的。

同时，移动的动画总是以默认位置为参照物为基础的。例如，在示例 13 中，已经定义 div 元素 left:200px，如果在 animate()方法中设置 left: "+100px"，则 div 元素并不是向右移动，而是向左移动 100 像素。对于 left: "-100px"移动动画来说，则会在现在固定位置基础上，向左移动 300 像素。

animate()方法的功能是很强大的，可以把第 2 个及其后面的所有参数都放置在一个对象中，在这个集合对象中包含动画选项的值，然后把这个对象作为第 2 个参数传递给 animate()方法。该参数可以包含下面多个选项。

- ☑ duration: 指定动画演示的持续时间，该选项与在 animate()方法中直接传递时间的作用是相同的。duration 选项也可以包含 3 个预定义的字符串，如 slow、normal 和 fast。
- ☑ easing: 该选项接收要使用的擦除效果的名称，需要插件支持。默认值为 swing。
- ☑ complete: 指定动画完成时执行的函数。
- ☑ step: 表示动画演示之后回调值。
- ☑ queue: 表示是否将使此动画进入动画队列。默认值为 true。

【示例 14】 设置一个动画队列，其中设置第 1 个动画不在队列中运行，此时可以看到第 1 个动画的字体变大和第 2 个动画的元素高度增加是同步进行的。当这两个动画同步进行完成之后，才触发第 3 个动画。在第 3 个动画中，设置 div 元素的最终不透明度为 0，则经过 2000 毫秒的淡出演示过程之后，该 div 元素消失。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").animate(           //第 1 个动画

```



```

        {height:"120%"},
        {duration: 5000, queue: false}
    ).animate({                //第 2 个动画, 将与第 1 个动画并列进行
        fontSize: "10em"
    },1000).animate({        //第 3 个动画
        opacity: 0
    }, 2000);
    })
}
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="自定义动画" />
<div style="border:solid 1px red;">自定义动画</div>
</body>
</html>

```

7.5.3 动态定位

通过 `animate()` 方法, 不仅能够控制其他效果方法中使用的样式属性, 还可以控制其他属性, 如 `left` 和 `top`。通过使用额外的属性, 能够创建更加精致、新颖的效果。例如, 可以在一个元素的高度增加到 50 像素的同时, 将它从页面的左侧移动到页面右侧。

在使用 `animate()` 方法时, 必须明确 CSS 对要改变的元素所施加的限制。例如, 在元素的 CSS 定位没有设置成 `relative` 或 `absolute` 的情况下, 调整 `left` 属性对于匹配的元素毫无作用。所有块级元素默认的 CSS 定位属性都是 `static`, 这个值精确地表明: 在改变元素的定位属性之前试图移动它们, 它们只会保持静止不动。

【示例 15】 设置一个动画队列, 在 2 秒中向右下角移动图像, 同时渐变不透明度为 50%, 动画完成后, 将执行回调函数, 提示动画完成的提示信息。演示效果如图 7.15 所示。注意, 当清除 `<p>` 标签中的 `position: relative` 声明之后, 整个动画将显示为无效。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").click(function(){
        $("p").animate({
            left:200,
            top:200,
            opacity: .5
        }, 2000, "linear", function(){alert("动画完成");});
    })
}
</script>
<title>上机练习</title>
</head>
<body>
<button>控制按钮 1</button>

```

```
<p style="position:relative"></p>
</body>
</html>
```

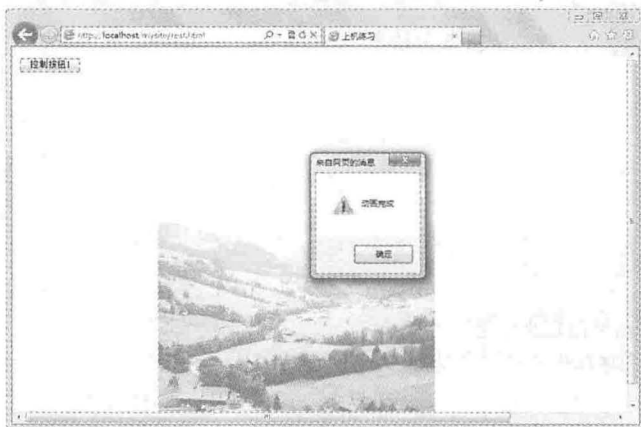


图 7.15 自定义效果

7.5.4 停止动画

jQuery 定义了 `stop()` 方法，该方法可以随时停止所有在指定元素上正在运行的动画。具体用法如下：

`stop([clearQueue], [jumpToEnd])`

- ☑ 参数 `clearQueue` 是一个布尔值，指示是否取消以列队动画。默认值为 `false`。
- ☑ 参数 `jumpToEnd` 是一个布尔值，指示当前动画是否立即完成。默认值为 `false`。

当一个元素调用 `stop()` 方法之后，当前正在运行的动画（如果有）立即停止。需要注意以下问题：

- ☑ 如果一个元素用 `slideUp()` 隐藏时，`stop()` 方法被调用，元素现在仍然被显示，但将是先前高度的一部分。不调用回调函数。
- ☑ 如果同一元素调用多个动画方法，后来的动画被放置在元素的效果队列中。这些动画不会开始，直到第 1 个动画完成。当调用 `stop()` 方法时，队列中的下一个动画立即开始。如果 `clearQueue` 参数提供 `true` 值，那么在队列中的其余动画被删除并永远不会运行。
- ☑ 如果 `jumpToEnd` 参数提供 `true` 值，当前动画将停止，但该元素会立即给予每个 CSS 属性的目标值。上面的 `slideUp()` 为例，该元素将立即隐藏。如果提供回调函数将立即被调用。当需要对元素做 `mouseenter` 和 `mouseleave` 动画时，`stop()` 方法明显是有效的。

【示例 16】当单击“自定义动画”按钮时，可以随时单击“停止动画”按钮停止动画的演示，如图 7.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").eq(0).click(function(){
        $("div").animate({
            fontSize : "10em"
        }, 8000);
    });
});
```

```

    $("input").eq(1).click(function(){
        $("div").stop();
    })
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="自定义动画" /><input type="button" value="停止动画" />
<div style="border:solid 1px red;">自定义动画</div>
</body>
</html>

```



图 7.16 控制动画

stop()方法包含两个可选的参数。第 1 个参数表示布尔值,如果设置为 true,则清空队列,立即结束所有动画。如果设置为 false,则如果动画队列中有等待执行的动画,会立即执行队列后面的动画。

第 2 个参数也是一个布尔值,如果设置为 true,则会让当前正在执行的动画立即完成,并且重设 show 和 hide 的原始样式,调用回调函数等。

7.5.5 关闭动画

jQuery 除定义了 stop()方法外,还定义了 off 属性。当这个属性设置为 true 时,调用时所有动画方法将立即设置元素为它们的最终状态,而不是显示效果。该属性解决了 jQuery 动画存在的以下几个问题:

- ☑ jQuery 被用在低资源设备。
- ☑ 动画使用户遇到可访问性问题。
- ☑ 动画可以通过设置这个属性为 false 重新打开。

【示例 17】 首先调用 jQuery.fx 空间下的属性 off,设置该属性值为 true,即关闭当前页面中所有的 jQuery,因此下面按钮所绑定的 jQuery 动画也是无效的。当单击按钮时,直接显示 animate()方法的第 1 个参数设置的最终样式效果。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    jQuery.fx.off = true;

```



```

    $("input").click(function(){
        $("div").animate({
            fontSize : "10em"
        }, 8000);
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="自定义动画" />
<div style="border:solid 1px red;">自定义动画</div>
</body>
</html>

```

关闭 jQuery 动画，对于配置比较低的电脑，或者遇到了可访问性问题，是非常有帮助的。如果要重新开启所有动画，只需要设置 jQuery.fx.off 属性值为 false 即可。

7.5.6 设置动画频率

jQuery 定义了 interval 属性，该属性可以设置动画的频率，以毫秒为单位。jQuery 动画默认是 13 毫秒。修改 jQuery.fx.interval 属性值为一个较小的数字可使动画在更快的浏览器中运行得更流畅，如 Chrome，但这样做有可能影响性能。

【示例 18】 修改 jQuery 动画的帧频为 100，则会看到更加精细的动画效果。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    jQuery.fx.interval = 100;
    $("input").click(function(){
        $("div").toggle( 3000 );
    });
})
</script>
<style type="text/css">
div { width:50px; height:30px; margin:5px; float:left; background:green; }
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="运行动画"/>
<div></div>
</body>
</html>

```

7.5.7 延迟动画

Delay()方法能够延迟动画的执行, 具体用法如下:

`delay(duration, [queueName])`

☑ 参数 `duration` 是一个用于设定队列推迟执行的时间, 以毫秒为单位的整数。

☑ 参数 `queueName` 是一个作为队列名的字符串, 默认是动画队列 `fx`。

`delay()`方法允许将队列中的函数延时执行。它既可以推迟动画队列中函数的执行, 也可以用于自定义队列延时时间是以毫秒为单位的, 数值越大, 动画越慢, 不是越快。字符串 `fast` 和 `slow` 分别代表 200 和 600 毫秒的延时。例如, 在 `<div id="foo">` 的 `slideUp()` 和 `fadeIn()` 动画之间设置 800 毫秒的延时:

`$('#foo').slideUp(300).delay(800).fadeIn(400);`

当这句语句执行时, 这个元素会以 300 毫秒卷起动画, 然后在 400 毫秒淡入动画前暂停 800 毫秒。

`jQuery.delay()` 用来在 jQuery 动画效果和类似队列中是最好的, 但不是替代 JavaScript 原生的 `setTimeout` 函数, 后者更适用于通常情况。

7.6 动画队列

jQuery 支持数据队列, 并通过定义 `queue()` 函数实现对队列的完整操作, 这对于一系列需要按次序执行的函数特别有用。例如, `animate` 动画、Ajax 异步请求和交互以及 `timeout` 等需要一定时间的函数。

实际上, jQuery 把队列看作是 `elem` 对象的数据缓存工具, 但是它与 `data()` 函数实现的数据缓存存在很大差异, 因为队列中存储的是将要被执行的一连串的动作函数。

7.6.1 添加动画队列

jQuery 定义了 `queue()` 工具函数, 该函数能够把函数加入队列, 这里的队列通常是一个函数数组。当为同一个元素设计连续动画时, 如多次执行 `animate()` 方法, jQuery 会自动将其加入名为 `fx` 的函数队列。但是, 如果需要对于多个元素依次执行动画, 就必须借助 `queue()` 函数手动设置队列。`queue()` 函数能够在匹配元素的队列最后添加一个函数, 并调用该函数。具体用法如下:

`jQuery.queue(element, queueName, newQueue)`

`jQuery.queue(element, queueName, callback())`

☑ 参数 `element` 是一个要附加队列函数的 DOM 元素, 或者是一个已附加队列函数 (数组) 的 DOM 元素。

☑ 参数 `queueName` 是一个含有队列名的字符串。默认是 `fx`, 标准的动画队列。

☑ 参数 `newQueue` 是一个替换当前函数队列内容的数组。

☑ 参数 `callback()` 为添加到队列的新函数。

每个元素可以通过 jQuery 包含一个或多个函数队列。在大多数应用中, 只有一个队列 (访问 `fx`) 被使用。队列允许一个元素来异步地访问一连串的动作, 而不终止程序执行。

`jQuery.queue()` 方法允许直接操纵这个函数队列, 用一个回调函数访问 `jQuery.queue()` 特别有用, 它让把新函数置入到队列的末端。

值得注意的是, 当使用 `jQuery.queue()` 添加一个函数时, 用户必须保证 `jQuery.dequeue()` 让下一个函数执行后被呼叫。

提示: 队列是一种特殊的线性列表结构, 它只允许在表的前端 (`front`) 进行删除操作, 在表的后端 (`rear`) 进行插入操作。允许插入操作的一端被称为队尾, 允许删除操作的一端称为队头。队列中没有元素时, 称

为空队列。在队列这种数据结构中，最先插入的元素必定最先被删除，最后插入的元素将最后被删除，因此队列又称为“先进先出”（FIFO——first in first out）的线性表。

【示例 19】 在按钮的 click 事件中定义了 6 个动作，其中第 3 个和第 5 个动作是通过 queue() 函数手动添加到队列中的。

但是，由于 queue() 函数是在队列末尾添加一个函数，则在该行后面的动作都将被忽视。所以，当在浏览器中预览时，小方块滑动到最右侧并调用末尾添加的队列函数之后，就停止了响应。演示效果如图 7.17 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $div = $("div");
    $("input").click(function(){
        $div.slideDown("slow");
        $div.animate({left:'+=400'},2000);
        $div.queue(function(){           //在队列的末尾添加一个函数
            $(this).addClass("bg");      //调用该回调函数之后动画将停止
        });
        $div.animate({left:'-=400'},2000);
        $div.queue(function(){
            $(this).removeClass("bg");
        });
        $div.slideUp("slow");
    });
});
</script>
<style type="text/css">
.bg { background:blue; }
div { position:absolute; width:50px; height:50px; background:red; left:0; top:50px; display:none; }
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="动画演示" />
<div></div>
</body>
</html>
```

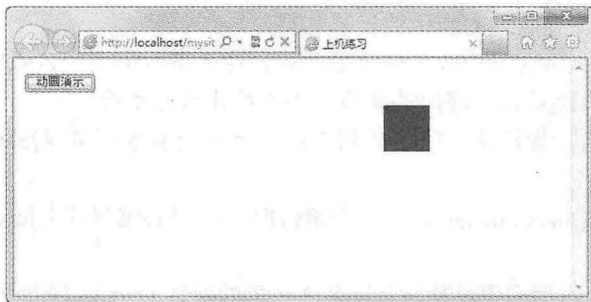


图 7.17 queue() 函数应用

7.6.2 显示动画队列

当为匹配的元素添加队列之后, 可以使用 `queue()` 函数获取对该队列的引用。具体用法如下:

`jQuery.queue(element, [queueName])`

☑ 参数 `element` 表示一个用于检查附加队列的 DOM 元素。

☑ 参数 `queueName` 表示一个含有队列名的字符串, 默认是 `fx`, 标准的动画队列。

这里的队列实际上就是一个函数数组, 并能够自动连续执行。参数 `name` 表示队列名称, 一般默认为 `fx`。

【示例 20】 获取 `div` 元素默认的 `fx` 队列, 并查询该队列中包含多少函数成员。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $div = $("div");
    $("input").click(function(){
        $div.slideDown("slow");
        $div.animate({left:'+=400'},2000);
        $div.animate({left:'-=400'},2000);
        $div.slideUp("slow");
        var x = $div.queue();           //获取 div 元素默认的队列 fx
        alert(x.length);               //显示 fx 队列包含的 4 个函数成员
    });
})
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="动画演示" />
<div></div>
</body>
</html>
```

如果匹配的元素不止一个, 则返回指向第 1 个匹配元素的队列, 即返回第 1 个元素包含的函数数组。

7.6.3 更新动画队列

一个队列执行完毕之后, 可以使用另一个队列进行替换, 具体实现方法是在 `queue()` 函数的第 2 个参数中传递一个队列, 将匹配元素的队列使用新的一个队列来代替, 即使用新的函数数组代替现在已执行的函数数组。

【示例 21】 分别为 `div` 元素设计两个动画序列, 其中第 1 个为默认的 `fx` 动画序列, 它直接被绑定在第 1 个按钮的 `click` 事件处理函数中。该动画序列包含 4 个动作函数, 按顺序作用于 `div` 元素, 分别为慢速显示、慢速前进、慢速后退和慢速隐藏元素。第 2 个动画序列通过 `queue()` 函数定义, 序列名称为 `fa`, 该序列中包含 4 个动作函数, 按顺序作用于 `div` 元素, 分别为快速显示、快速前进、快速后退和快速隐藏。然后使用 `queue()` 函数获取名称为 `fa` 的动画序列, 并调用 `queue()` 函数使用 `fa` 动画序列替换 `fx` 动画序列。演示效果如

图 7.18 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var $div = $("div");
    $("input").eq(0).click(function(){           //默认的第 1 个动画序列，慢速动画
        $div.slideDown("slow");
        $div.animate({left:'+=400'},4000);
        $div.animate({left:'-=400'},4000);
        $div.slideUp("slow");
    });
    $div.queue("fa",function(){                  //自定义动画序列，快速动画
        $div.slideDown("fast");
        $div.animate({left:'+=400'},200);
        $div.animate({left:'-=400'},200);
        $div.slideUp("fast ");
    });
    var fa = $div.queue("fa");                  //获取对自定义动画序列的引用
    $("input").eq(1).click(function(){
        $div.queue("fx",fa);                    //使用 fa 动画序列覆盖默认的 fx 动画序列
    });
});
</script>
<style type="text/css">
.bg { background:blue; }
div { position:absolute; width:50px; height:50px; background:red; left:0; top:50px; display:none; }
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="执行慢速演示" />
<input type="button" value="更新动画，执行快速演示" />
<div></div>
</body>
</html>
```



图 7.18 更新队列函数

注意，在动画序列执行过程中，并不是立即进行替换，而是等到当前正在执行的动作完成之后，才停止正在执行的 fx 序列，并继续执行第 2 个 fa 动画序列。

在 `queue(name, queue)` 方法中, 如果第 2 个参数是一个空数组 (`[]`), 则将会清除原来的动画序列。例如, 下面代码将清空匹配的 `div` 元素的默认动画序列:

```
$("#div").queue("fx", []);
```

7.6.4 删除动画队列


`dequeue()` 函数能够删除指定队列中最顶部的函数, 并执行这个队列函数。实际上, `dequeue()` 函数是将函数数组中的第 1 个函数取出来, 并执行这个函数。那么当再次执行 `dequeue` 时, 得到的是另一个函数。如果不执行 `dequeue`, 则队列中的下一个函数将永远不会执行。`dequeue()` 函数包含一个参数, 用来指定队列的名称, 默认为 `fx`。

【示例 22】 使用 `dequeue()` 函数结束自定义队列函数, 并使队列继续进行下去。这样动画将会连续播放, 直到最后一个函数被执行为止。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var $div = $("#div");
    $("input").click(function(){
        $div.slideDown("slow");
        $div.animate({left:'+=400'},2000);
        $div.queue(function(){
            $(this).addClass("bg");
            $(this).dequeue();
        });
        $div.animate({left:'-=400'},2000);
        $div.queue(function(){
            $(this).removeClass("bg");
            $(this).dequeue();           //删除最顶部的函数, 并继续执行队列
        });
        $div.slideUp("slow");
    });
});
</script>
<style type="text/css">
.bg { background:blue; }
div { position:absolute; width:50px; height:50px; background:red; left:0; top:50px; display:none; }
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="动画演示" />
<div></div>
<div></div>
</body>
</html>
```


第 8 章

工具函数

( 视频讲解：1 小时 21 分钟)

jQuery 是基于跨浏览器的技术框架，也就是说，当使用 jQuery 代码时，不用为了兼容不同浏览器而烦恼。jQuery 方法都是针对 jQuery 对象的，即都是用来操作 \$() 函数包装的一组 DOM 元素，当然 jQuery 也定义了很多工具函数，这些函数的命名空间为 \$，但不操作包装集，用户可以把它看作是顶层函数，不同之处是它们定义在 \$ 实例上，而不是定义在 window 实例上，类似于静态类型函数。

通常来说，工具函数的主要任务是操作除 DOM 元素以外的 JavaScript 对象，或者执行一些非对象相关的操作。本章将介绍大多数 \$ 级别的实用工具函数以及几个有用的标志，以方便读者更深刻地认识和理解这些工具函数及其使用方法。

8.1 jQuery 标志

jQuery 通过定义在 \$ 上的变量为开发人员提供一些有用的客户信息，通过这些标志信息可以方便地检测当前浏览器的功能，以便用户基于这些信息进行决策。这些标志信息包括 jQuery.browser、jQuery.boxModel 和 jQuery.support。

8.1.1 检测用户代理

浏览器检测的方法有许多种，如字符串检测法和特征检测法。

字符串检测法就是根据 navigator.userAgent 属性返回值进行检测。但是，jQuery 从 1.3 版本开始就不再支持使用这种方法，原因就是它使用比较麻烦，与 jQuery 技术框架的灵巧特色相违背。

特征检测法就是根据浏览器是否支持特定功能来决定操作的方式。特征检测法是一种非精确的检测方法，也是最安全的检测途径。因为准确检测浏览器的类型和型号是一件很困难或者说很麻烦的事情，而且很容易存在误差。如果不关心浏览器的身份，仅在意浏览器的执行能力，那么使用特征检测法就完全可以满足需要。例如：

```
if(document.getElementsByName){           //如果存在，getElementsByName 则使用该方法获取 a 元素
    var a = document.getElementsByName("a");
}
else(document.getElementsByTagName){       //如果存在，getElementsByTagName 则使用该方法获取 a 元素
    var a = document.getElementsByTagName("a");
}
```

当使用对象、方法或属性时，可以先检测当前浏览器是否支持它。在逻辑表达式中，如果浏览器支持，则会返回该对象、属性或方法，这时 JavaScript 就会强制把这些对象或成员转换为 true。如果不支持，则会返回 undefined，JavaScript 会自动把它转换为布尔值 false。

如果要检查方法或函数时，应注意不要附加小括号运算符；否则 JavaScript 解释器会调用该方法或函数，同时如果指定函数或方法不存在，就会产生编译错误。

检测浏览器的类型主要通过 Navigator 对象的 userAgent 属性来实现，即通过引用 Window 对象的 navigator 属性来读取，具体用法如下：

```
var browser = navigator.userAgent;
```

jQuery 定义了 browser 属性，通过该属性可以获取当前浏览器的类型，浏览器对象检测技术与此属性共同使用可提供可靠的浏览器检测支持。browser 属性允许检测 4 个最流行的浏览器类，如 Internet Explorer、Mozilla、Webkit 和 Opera，以及每个版本信息标志。可用的标志包括 webkit (as of jQuery 1.4)、safari (deprecated)、opera、msie 和 mozilla。

【示例 1】 通过遍历对象属性，获取每个属性值，并确定当前浏览器的类型。演示效果如图 8.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var browser = $.browser;
    var temp = ""
    for(var name in browser){
        if(browser[name] == true)
            temp += name + " = " + browser[name] + ", <strong>当前浏览器是 " + name + " </strong><br />";
        else
            temp += name + " = " + browser[name] + "<br />";
    }
    $("div").html(temp)
})
</script>
<title>上机练习</title>
</head>
<body>
<div></div>
</body>
</html>
```



图 8.1 获取浏览器的类型

通过示例 1 可以直接调用这些属性来检测当前浏览器是否为特定类型浏览器。这些属性在 DOM 树加载完成前有效，因此可用于为特定浏览器设置 ready 事件。

【示例 2】 分别为不同浏览器编写不同的页面初始化配置函数。访问方式可以通过点号运算符直接调用属性，也可以作为名称下标进行访问。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
if($.browser.msie){
    $(function(){
        alert("IE 浏览器专用页面初始化函数！");
    })
}
else if($.browser.safari){
    $(function(){
        alert("Safari 浏览器专用页面初始化函数！");
    })
}
else if($.browser["opera"]){
    $(function(){
        alert("Opera 浏览器专用页面初始化函数！");
    })
}
else if($.browser["mozilla"]){
    $(function(){
        alert("Firefox 浏览器专用页面初始化函数！");
    })
}
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

browser 使用 navigator.userAgent 来确定平台，它很容易被用户或浏览器本身的失实陈述欺骗。而 support 属性比 browser 提供更有效的检测特定功能的支持，因此不推荐使用这个属性，可以尝试使用功能检测来代替（jQuery.support）。

分析源代码，jQuery 正是通过下面代码计算当前浏览器的类型的：

```
//获取用户代理字符串信息，并把字符串全部转换为小写形式
var userAgent = navigator.userAgent.toLowerCase();
//定义 jQuery 全局对象，该对象包含 5 个公共属性
jQuery.browser = {
    //匹配并过滤出字符串中的版本号
    version: (userAgent.match( /.+?(?:rv|it|ra|ie)[V: ]([\d.]+) / ) || [0,0])[1],
    safari: /webkit/.test( userAgent ),           //匹配 webkit 关键字
    opera: /opera/.test( userAgent ),             //匹配 opera 关键字
    msie: /msie/.test( userAgent ) && !/opera/.test( userAgent ), //匹配 msie 关键字
```



```
//匹配 mozilla 关键字
mozilla: /mozilla/.test( userAgent ) && !/(compatible|webkit)/.test( userAgent )
};
```

可以在上面代码基础上对其进行封装和完善。例如，在下面示例中，定义了 `browser()` 全局函数，通过向该函数传递一个字符串，以测试当前浏览器是否是指定类型的浏览器。

```
<script type="text/javascript">
//参数: str 是一个字符串类型的参数, 包含下面几个固定值。
//ie: 匹配 IE 浏览器。
//op: 匹配 Opera 浏览器。
//sa: 匹配 Safari 浏览器。
//ch: 匹配 Chrome 浏览器。
//ff: 匹配 Firefox 浏览器。
//返回值: 布尔值, true 表示是特定类型的浏览器, false 表示不是特定类型的浏览器
function browser(str){
    var userAgent = navigator.userAgent.toLowerCase();
    switch (str){
        case "ie":
            return /msie/.test( userAgent ) && !/opera/.test( userAgent );
        case "ff":
            return /mozilla/.test( userAgent ) && !/(compatible|webkit)/.test( userAgent );
        case "sa":
            return /webkit/.test( userAgent );
        case "op":
            return /opera/.test( userAgent );
        case "ch":
            return /chrome/.test( userAgent );
        default:
            return false;
    }
}
//测试当前浏览器类型, 使用一个长表达式语句来模拟多条件的判断结构
window.onload = function(){
    browser("ie") && (useragent = "当前浏览器是 IE 浏览器") ||
    browser("ff") && (useragent = "当前浏览器是 Firefox 浏览器") ||
    browser("sa") && (useragent = "当前浏览器是 safari 浏览器") ||
    browser("op") && (useragent = "当前浏览器是 Opera 浏览器") ||
    browser("ch") && (useragent = "当前浏览器是 Chrome 浏览器");
    alert(useragent);
}
</script>
```

8.1.2 检测版本号

在 jQuery 中可以借助 `jQuery.browser.version` 属性获取浏览器的版本号。例如，下面代码可以返回当前浏览器的版本号，返回值是字符串类型：

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    alert( $.browser.version );
})
</script>
```

8.1.3 检测盒模型

通过 `jQuery.boxModel` 标志可以获取当前页面使用的是哪一种盒模型。如果页面使用符合 W3C 标准的盒模型，则该属性返回值 `true`。如果页面使用 IE 浏览器的盒模型，则返回 `false`。

盒模型就是浏览器解析元素的方法，它是 CSS 布局中一个基本概念。盒模型规定了如何决定元素（与其内边距和边框间距）的内容大小，外边距虽然也是盒模型的一部分，但是不参与确定内容的大小。

除 IE 浏览器外，其他浏览器都支持 W3C 标准的盒模型，而 IE 浏览器能够根据页面模式（严格模式或者怪异模式）有选择地使用不同类型的盒模型。如果页面顶部声明了文档类型（DOCTYPE），则 IE 也会采用严格模式，即 W3C 标准的盒模型解析元素。如果文档中没有包含文档类型（DOCTYPE），或者包含了无法识别的文档类型声明，则会以怪异模式显示，并按 IE 传统的盒模型来解析元素。

总之，IE 的传统盒模型与 W3C 盒模型的主要区别在于如何解释元素的 `width` 和 `height` 属性的计算标准。请记住以下两条原则。

- ☑ IE 传统盒模型：`width` 和 `height` 属性包含内边距和边框宽度。
- ☑ W3C 盒模型：`width` 和 `height` 属性不包含内边距和边框宽度。

如果直接使用 JavaScript 实现检测浏览器在解析当前文档时所采用的解析模型，可以利用脚本在当前文档中创建一个 `div` 元素，设置 `div` 元素的左侧补白和边框宽度为 1 个像素，然后调用该元素的 `offsetWidth` 属性获取 `div` 元素的可见宽度。如果它的值等于 2 像素，则说明是按 W3C 标准进行解析的，否则就是按照 IE 的怪异模型进行解析的。实现的详细代码如下：

```
<script type="text/javascript">
function isBoxModel(){
    var div = document.createElement("div");
    div.style.width = div.style.paddingLeft = "1px";
    document.body.appendChild( div );
    var width = div.offsetWidth;
    div.style.display = 'none';
    document.body.removeChild( div )
    return width === 2;
}
window.onload = function(){
    alert( $.boxModel  &&  "支持 W3C 标准盒模型"  ||  "支持 IE 的怪异解析模式" );
}
</script>
```

但是，通过调用 `jQuery.boxModel` 属性可以快速确定浏览器在解析当前文档是否支持 W3C 标准的盒模型。如果返回值为 `true`，则表示支持，否则表示不支持，即支持 IE 的怪异模式。例如，下面代码可以感性地认识该属性的应用：

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    alert( $.boxModel  &&  "支持 W3C 标准盒模型"  ||  "支持 IE 的怪异解析模式" );
})
</script>
```

在 jQuery 1.3 中不建议使用，如果要检测当前页面中浏览器是否使用标准盒模型渲染页面，建议使用 `jQuery.support.boxModel` 属性来代替。

8.1.4 检测功能或缺陷

jQuery 定义了 `support` 属性，该属性返回一个 `Object` 对象，在该对象中包含了一组属性，它们代表了不

同的浏览器功能或缺陷的存在的集合。

使用 `support` 执行功能检测是一个很好的做法，而不用 `$.browser` 来检测当前用户代理，所有这些支持的属性值都通过特性检测（和不使用任何浏览器的形式）。虽然 `jQuery` 包含了一些属性，开发者可以添加自己需要的。许多 `jQuery.support` 性能相当低，所以对插件和 `jQuery` 核心非常有用，而不是一般日常开发。`support` 对象包含的属性及其测试内容说明如表 8.1 所示。

表 8.1 jQuery.support 对象包含的属性及其测试内容说明

属 性	说 明
<code>boxModel</code>	如果浏览器解析当前文档是以 W3C CSS 盒模型来渲染的，则返回 <code>true</code> ；如果是在 IE 6 和 IE 7 的怪异模式中，则返回 <code>false</code> ；在页面初始化之前，则返回 <code>null</code>
<code>cssFloat</code>	如果浏览器使用 <code>cssFloat</code> 属性来访问 CSS 的 <code>float</code> 样式值，则返回 <code>true</code> ，否则返回 <code>false</code> 。在 IE 浏览器中会返回 <code>false</code> ，因为它使用 <code>styleFloat</code> 属性来访问 CSS 的 <code>float</code> 样式值
<code>hrefNormalized</code>	如果浏览器从 <code>getAttribute("href")</code> 返回的是原封不动的结果，则返回 <code>true</code> ，否则返回 <code>false</code> 。在 IE 浏览器中会返回 <code>false</code> ，因为它对返回的结果进行了格式化处理
<code>htmlSerialize</code>	如果浏览器通过 <code>innerHTML</code> 插入 <code>a</code> 元素时，会自动序列化这些超链接，则返回 <code>true</code> ，否则返回 <code>false</code> 。目前，在 IE 浏览器中会返回 <code>false</code>
<code>leadingWhitespace</code>	如果浏览器在使用 <code>innerHTML</code> 时，保持前导空白字符，则返回 <code>true</code> ，否则返回 <code>false</code> 。目前在 IE 6~IE 8 版本浏览器中会返回 <code>false</code>
<code>noCloneEvent</code>	如果浏览器在克隆元素时不会连同事件处理函数一起复制，则返回 <code>true</code> 。目前，在 IE 中返回 <code>false</code>
<code>objectAll</code>	如果在某个元素对象上执行 <code>getElementsByName("*")</code> 会返回所有子孙元素，则为 <code>true</code> 。目前，在 IE 7 中为 <code>false</code>
<code>opacity</code>	如果浏览器能适当解释透明度样式属性，则返回 <code>true</code> ，由于 IE 浏览器使用 <code>alpha</code> 滤镜实现，故返回 <code>false</code>
<code>scriptEval</code>	使用 <code>appendChild()</code> 或 <code>createTextNode()</code> 方法插入脚本代码时，浏览器是否执行脚本。目前，在 IE 中不能够执行，故返回 <code>false</code> 。IE 使用 <code>text</code> 方法插入脚本代码可以执行
<code>style</code>	如果 <code>getAttribute("style")</code> 返回元素的行内样式，则为 <code>true</code> 。由于 IE 使用 <code>cssText</code> 返回元素的行内样式，故返回 <code>false</code>
<code>tbody</code>	如果浏览器允许 <code>table</code> 元素不包含 <code>tbody</code> 元素，则返回 <code>true</code> 。目前在 IE 中会返回 <code>false</code> ，它会自动插入缺失的 <code>tbody</code> 元素

所有这些支持的属性值都通过特性检测来实现，而不是用任何浏览器检测。下面这些非常棒的资源都是用于解释这些特性检测是如何工作的：

- ☑ <http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting>
- ☑ <http://yura.thinkweb2.com/cft/>
- ☑ http://www.jibbering.com/faq/faq_notes/not_browser_detect.html

8.2 兼容 JavaScript 库

`jQuery` 在 1.3.6 版本开始就引入了 JavaScript 库兼容机制，即在同一个页面可以同时使用多个 JavaScript 库。一般来说，`$` 变量最容易发生冲突，因为不同 JavaScript 库中都使用了 `$` 标识符，但是它们在不同的库中所表示的语义是不同的。例如，`jQuery` 定义 `$` 符号代表 `jQuery` 对象，而 `Prototype` 技术库也引用了 `$` 命名空间。如果把它们直接导入到同一个文档中，可能就会引发命名空间的混乱，为此 `jQuery` 提供了 `$.noConflict()` 实用工具函数，该函数能够放弃对 `$` 变量的控制，让给其他库或者脚本使用。具体用法如下：


```
jQuery.noConflict([removeAll])
```

removeAll 判断是否从全局范围内去除所有 jQuery 变量的布尔值，包括 jQuery 本身。

因为\$只不过是 jQuery 的别名，所以在应用 jQuery.noConflict()函数之后，jQuery 的全部功能依然可以使用，但是此时只能使用 jQuery，而不是\$。

【示例 3】 为了方便理解这个工具的应用，可以输入下面代码进行测试：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
var $ = function(){
    alert("其他库别名");
}
</script>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    alert("jQuery 库别名");
})
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

在示例 3 中，先于 jQuery 库之前命名的一个\$变量，为该变量定义一个简单的函数，然后导入 jQuery 库，再调用\$()函数，就可以看到浏览器根据最后导入的 jQuery 库的命名空间来执行\$()函数，如图 8.2 所示。

如果希望执行 jQuery 库前面的\$()或者其他库的命名空间中的\$()函数，则只需要在导入 jQuery 库后的脚本中调用 jQuery.noConflict()函数即可。例如，针对示例 3，可以按如下方法来设计，则在浏览器中浏览时，可以看到最先定义的\$()函数有效，如图 8.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
var $ = function(){
    alert("其他库别名");
}
</script>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
jQuery.noConflict();    //恢复$变量
$(function(){
    alert("jQuery 库别名");
})
</script>
<style type="text/css"></style>
```

```
<title>上机练习</title>
</head>
<body>
</body>
</html>
```



图 8.2 应用 jQuery 库



图 8.3 恢复\$变量名

通过这种方式可以确保 jQuery 不会与其他库的\$对象发生冲突,在运行 jQuery.noConflict()函数之后,就只能使用 jQuery 变量访问 jQuery 对象。例如,在要用到\$()的地方,就必须换成 jQuery()。

注意, noConflict()函数必须在导入 jQuery 库之后,并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前,除非 jQuery 是最后一个导入的。分析 noConflict()函数的源代码,可以看到 noConflict()函数实际上是把备份的\$变量进行恢复,恢复到最初的状态。

```
noConflict: function( deep ) {
    window.$ = _$;
    if ( deep )
        window.jQuery = _jQuery;
    return jQuery;
},
```

如果 jQuery 命名空间也发生了冲突,可以使用 jQuery.noConflict(deep)函数进行解决,它是 noConflict()函数的高级版本。当参数 deep 为 true 时,该函数能够把\$和 jQuery 的控制权都交还给原来的库,因此将完全重新定义 jQuery。

【示例 4】继续以上面示例为基础,现在调用 jQuery.noConflict()函数,并向其传递一个 true 参数,则 jQuery 会使用内部变量 jQuery 恢复 jQuery 库之前的最初功能。在下面示例中,定义全局变量 jQuerySelf 暂存 jQuery 名字空间,并通过 jQuery.noConflict(true)函数恢复 jQuery 最初的命名空间语义。所以,在下面示例中 will 看到如何避免库冲突,同时又能够实现库之间相安无事,可以在同一个文档中交叉使用。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script type="text/javascript">
var $ = function(){
    alert("其他库别名");
}
</script>
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
//把 jQuery 库另存为 jQuerySelf
var jQuerySelf = jQuery.noConflict(true);
jQuery(function(){ //将执行其他库名字空间
```

```

    alert("jQuery 库名");
  })
  jQuerySelf(function(){    //将执行 jQuery 库名字空间
    alert("jQuery 库名");
  })
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

jQuery.noConflict(deep)函数的源代码可以参阅 jQuery.noConflict()函数,其实现原理很简单,即如果参数值为 true,则使用临时变量_jQuery 恢复它的最初功能。

8.3 对象和集合操作

jQuery 定义了很多实用工具函数,用来操作除 DOM 元素以外的 JavaScript 对象。一般来说,被设计用来操作 DOM 元素的功能是作为 jQuery 命令提供的。虽然一些工具函数能够用来操作 DOM 元素,这些 DOM 元素本质上也是 JavaScript 对象,但是工具函数的主要任务不是针对 DOM 元素,而是额外完成一些 JavaScript 功能。

8.3.1 处理字符串

JavaScript 提供了很多原生的字符串处理方法,但是没有定义修剪字符串的方法,而这些方法在实际开发中是非常实用的。例如,在处理表单提交的数据中,经常需要清理掉字符串前后的空白。由于这些空白依然是合法的字符,所以很容易会影响程序的后期处理。

jQuery 定义了 trim()函数能够清理字符串前后的空白。trim()是一个全局函数,可以直接使用 jQuery 对象进行调用。该方法包含一个字符串型的参数,即将被修剪的字符串,返回修剪后的字符串。具体用法如下:

```
jQuery.trim(str)
```

参数 str 表示修剪的字符串。

【示例 5】演示字符串在被 jQuery 的 trim()修剪前后的字符串长度变化,修剪前字符串长度为 15,修剪后为 5。演示效果如图 8.4 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("button").click(function () {
        var str = "      修剪字符串      ";
        alert(str.length);
        str = jQuery.trim(str);
        alert(str.length);
    });
});

```



```

})
</script>
<title>上机练习</title>
</head>
<body>
<button>修剪字符串</button>
</body>
</html>

```



图 8.4 修剪字符串

jQuery 定义的 trim() 函数使用 JavaScript 实现也很简单，只需要借助正则表达式即可。例如，下面示例定义了一个 trim() 全局函数，并调用该函数修剪字符串。代码如下：

```

<script type="text/javascript">
function trim(text){
    return (text || "").replace( /\s+|\s+$/g, "" );
}
window.onload = function(){
    var str = "   去掉字符串起始和结尾的空格   ";
    alert(str.length);    //返回 19
    str = trim(str);
    alert(str.length);    //返回 13
}
</script>

```

8.3.2 把对象转换为字符串

jQuery 定义了 param() 函数，它能够将字符串创建为一个序列化的数组或对象，返回值为有序化的字符串。该方法特别适用于一个 URL 地址查询字符串或 Ajax 请求。具体用法如下：

```

jQuery.param(obj)
jQuery.param(obj, traditional)

```

☑ 参数 obj 为一个数组或序列化的对象。

☑ 参数 traditional 表示一个布尔值，设置是否执行了传统的 shallow 的序列化。

提示，param() 函数是 serialize() 方法的基础。所谓序列化，就是数组或者 jQuery 对象按照名/值 (name/value) 对格式进行序列化，而 JavaScript 普通对象按照 key/value 对格式进行序列化。

【示例 6】param() 函数能够把列表结构的对象 obj 转换为字符串类型的名/值对字符串，返回字符串 “user=zhangsan&pass=12345678”。演示效果如图 8.5 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

```

```

<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("button").click(function () {
        var option = {
            user:"zhangsan",
            pass:12345678
        };
        var str = jQuery.param( option );
        $("#result").text(str);
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<button>定义序列化字符串</button>
<div id="result"></div>
</body>
</html>

```



图 8.5 序列化字符串

8.3.3 判断数组类型

数组和对象都是散列式列表结构，它们都可以存储大量数据，开发人员喜欢使用数组或者对象来进行数据周转。但是数组和对象的操作方法各异，如何在开发中快速了解当前值是数组或者是对象就显得非常重要。isArray()函数是jQuery定义的负责检测对象是否为数组的专用工具，它可以快速判断指定对象是否为数组。具体用法如下：

```
jQuery.isArray(obj)
```

参数 obj 表示一个用来测试是否为一个数组的对象，返回值为布尔值。如果确定参数是一个数组，则返回 true，否则返回 false。

可以使用该函数检测对象是否是一个 JavaScript 数组，而不是伪数组，即类似数组的对象，如 jQuery 对象。

【示例 7】 检测变量 a 是否为数组，如果是数组则弹出提示信息。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >

```

```

$(function(){
    var a = [
        {width:400},
        {height:300}
    ];
    if(jQuery.isArray( a ))
        alert("变量 a 是数组");
})
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

jQuery 定义的 isArray() 函数实现起来很简单, 直接调用该对象的 toString() 方法即可。考虑到参数对象的 toString() 方法可能会被重写, 因此, 直接调用 Object 对象的原型方法即可。使用 JavaScript 实现的代码如下:

```

<script type="text/javascript">
function isArray( obj ){
    return Object.prototype.toString.call(obj) === "[object Array]";
}
</script>

```

8.3.4 判断函数类型

isFunction() 函数是 jQuery 定义的用来检测指定对象是否为函数类型的函数。该函数与 isArray() 函数用法相同, 具体用法如下:

```
jQuery.isFunction(obj)
```

参数 obj 表示用于测试是否为函数的对象。

【示例 8】 调用 each() 工具函数遍历 objs 数组, 检测每个数组元素的数据类型。如果是函数, 则显示 true 信息, 否则显示 false 信息。演示效果如图 8.6 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    function stub() {}
    var objs = [
        function () {},
        { x:15, y:20 },
        null,
        stub,
        "function"
    ];
    jQuery.each(objs, function (i) {
        var isFunc = jQuery.isFunction(objs[i]);
        $("span").eq(i).text(isFunc);
    });
});

```



```

    });
  })
</script>
<style type="text/css">
div { color:blue; margin:2px; font-size:14px; }
span { color:red; }
</style>
<title>上机练习</title>
</head>
<body>
<pre>
    var objs = [
        function () {},
        { x:15, y:20 },
        null,
        stub,
        "function"
    ];
</pre>
<div>jQuery.isFunction(objs[0]) = <span></span></div>
<div>jQuery.isFunction(objs[1]) = <span></span></div>
<div>jQuery.isFunction(objs[2]) = <span></span></div>
<div>jQuery.isFunction(objs[3]) = <span></span></div>
<div>jQuery.isFunction(objs[4]) = <span></span></div>
</body>
</html>

```



图 8.6 判断函数类型

在 jQuery 1.3 版本以后, IE 浏览器提供的函数, 如 `alert` 和 `getAttribute` (DOM 元素方法) 将不被认为是函数。isFunction() 函数可以直接使用 JavaScript 代码实现:

```

function isFunction( obj ){
    return Object.prototype.toString.call(obj) === "[object Function]";
}

```

8.3.5 判断特殊对象

jQuery 根据开发需要, 不断增加了一些特定需求的判断函数, 这些函数能够检测特定形式的对象。简单说明如下。

1. 检测空对象

jQuery.isEmptyObject() 函数能够检测对象是否为空, 即对象不包含任何属性。具体用法如下:

```
jQuery.isEmptyObject(object)
```

参数 object 表示要检查的对象，返回值为布尔值。如果是空对象则返回 true，否则返回 false。

在 jQuery 1.4 中，该方法既检测对象本身的属性，也检测从原型继承的属性，因此没有使用 hasOwnProperty。

2. 检测纯对象

jQuery.isPlainObject() 能够检测一个对象是否为纯对象，即通过对象直接量 ({}) 或者 Object 构造函数 (new Object) 创建的对象。具体用法如下：

```
jQuery.isPlainObject(object)
```

参数 object 表示要检查的对象，返回值为布尔值。如果是纯粹对象则返回 true，否则返回 false。

3. 检测 Window 对象

jQuery.isWindow() 能够检测一个对象是否为 Window 对象，即检测一个对象是否为一个窗口。具体用法如下：

```
jQuery.isWindow(obj)
```

参数 object 表示要检查的对象，返回值为布尔值。如果是窗口对象则返回 true，否则返回 false。

使用该函数可以确定当前操作是否为一个浏览器窗口操作，如当前窗口或一个 iframe。

4. 检测 XML 文档

jQuery.isXMLDoc() 能够检测一个 DOM 节点是否在 XML 文档中 (或者是一个 XML 文档)。具体用法如下：

```
jQuery.isXMLDoc( node )
```

参数 node 表示要检查的节点对象，返回值为布尔值。如果是 XML 文档中的节点对象则返回 true，否则返回 false。

8.3.6 对数组和集合进行迭代

不管是数组，还是对象，JavaScript 都提供方法对它们进行迭代。如果是数组，则可以使用 for 循环遍历数组元素；如果是对象，则可以利用 for in 循环将对象的属性进行迭代。尽管这种用法比较简单，但是在频繁的迭代操作中，这种繁琐的操作有时候会让人很头疼。jQuery 简化了这种操作，把所有工作都交付给 each() 函数实现。

each() 函数是个通用的迭代函数，它可以用来无缝迭代对象和数组。数组和类似数组的对象通过一个长度属性 (如一个函数的参数对象) 来迭代数字索引，从 0 到 length -1。其他对象迭代通过其命名属性。具体用法如下：

```
jQuery.each(collection, callback(indexInArray, valueOfElement))
```

☑ 参数 collection 表示遍历的对象或数组。

☑ 参数 callback(indexInArray, valueOfElement) 表示每个成员/元素执行的回调函数。该函数将在遍历每个成员时触发。回调函数包含两个默认参数，第 1 个参数为对象成员或数组的索引，第 2 个参数为对应变量的内容。

\$.each() 函数和 each() 方法是不一样的。each() 方法专门用来遍历一个 jQuery 对象，而 \$.each() 函数可用于迭代任何集合，无论是“名/值”对象 (JavaScript 对象) 或阵列。在一个数组的情况下，回调函数每次传递一个数组索引和相应的数组值。(该值也可以通过 this 关键字进行访问，但是 JavaScript 将始终包裹 this 值作为一个 Object，即使它是一个简单的字符串或数字值。) 该方法返回其第 1 个参数，是迭代的对象。

【示例 9】调用 jQuery.each() 函数遍历数组 a，然后在遍历过程中，逐一提示该数组元素的下标值和元素值。演示效果如图 8.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var a = [
        {width:400},
        {height:300}
    ];
    jQuery.each(a,function(name,value){
        alert( name + " = " + value);
    })
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```



图 8.7 迭代对象成员

如果中途需要退出 each() 循环, 则可以在回调函数中返回 false, 其他返回值将被忽略。

【示例 10】 以示例 9 为基础, 在 each() 函数中添加一个条件语句, 如果数组下标超过 0, 则退出循环。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var a = [
        {width:400},
        {height:300}
    ];
    jQuery.each(a,function(name,value){
        if(name>0) return false;           //仅执行一次循环即退出
        alert( name + " = " + value);
    })
})
</script>
<style type="text/css"></style>

```



```

<title>上机练习</title>
</head>
<body>
</body>
</html>

```

注意, jQuery 的 each() 函数与 jQuery 对象的 each() 方法功能相同, 但是用法不同。另外, each() 函数可用于遍历任何对象。

8.3.7 生成数组

散列表结构的数据可能是数组类型, 也可能是对象类型。由于数组和对象类型拥有不同的操作方法, 特别是数组对象, JavaScript 为其定义了众多强大的处理方法。因此, 在 DOM 中经常需要把列表结构的数据转换为数组。

例如, 使用 jQuery 获取文档中所有 li 元素, 则返回的应该是一个类似数组结构的对象。但是如果直接为其调用 reverse() 数组方法, 则会显示编译错误, 因为 \$("li") 返回的是一个类数组结构的对象, 而不是数组类型数据。如果直接使用 document.getElementsByTagName("li") 获取 li 元素集合, 返回值也是一个类数组结构的对象, 该对象无法直接调用数组方法。

jQuery 的 makeArray() 函数能够把这些类数组结构的对象转换为真正的 JavaScript 数组。该方法的具体用法如下:

```
jQuery.makeArray( obj )
```

参数 obj 表示转换成一个原生数组的任何对象。转换后, 任何有特殊功能的对象将不再存在, 返回一个普通的数组。

一般来说, 无论是在 jQuery 中还是在 JavaScript 中很多方法都返回类似数组的对象。例如, jQuery 的构造函数 \$() 返回一个 jQuery 对象, 该对象具有许多的数组的属性, 如 length 属性、[] 数组访问运算符等。但和数组并不完全一样, 缺少一些对数组的内置方法, 如 pop() 和 reverse() 等。

【示例 11】 为了重新排序列表项顺序, 使列表项包含的数字按倒序排列, 先使用 makeArray() 函数把类数组对象转换为数组对象, 然后再为其调用 reverse() 方法。这时就可以看到页面中的列表结构被颠倒过来。演示效果如图 8.8 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var arr = jQuery.makeArray($("li"));           //转换为数组
    $("ul").html(arr.reverse());                 //调用 reverse()方法
})
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
<ul>
    <li>1</li>
    <li>2</li>

```

```

<li>3</li>
<li>4</li>
<li>5</li>
</ul>
</body>
</html>

```



图 8.8 makeArray()函数的应用

8.3.8 对数组进行筛选

遍历数组以便查找匹配的特定元素，是处理大量数据应用的频繁需求。当然用户也想要对数据进行筛选，查找在特定界限之上或者之下的项，或者匹配特定模式的项，为此 jQuery 定义了 `grep()` 函数，该函数能够根据过滤函数过滤掉数组中不符合条件的元素。具体用法如下：

```
jQuery.grep(array, function(elementOfArray, indexInArray), [ invert ])
```

- ☑ 参数 `array` 表示数组，用来搜索。
- ☑ 参数函数 `function(elementOfArray, indexInArray)` 用来处理每个项目的比对。第 1 个参数给函数是项目，第 2 个参数是索引。该函数应返回一个布尔值。`this` 将是全局的窗口对象。
- ☑ 参数 `invert` 如果为 `false`，或没有提供，函数返回一个所有元素组成的数组，对于 `callback` 返回 `true`。如果 `invert` 为 `true`，函数返回一个所有元素组成的数组，对于 `callback` 返回 `false`。

`$.grep()` 方法删除数组必要项目，以使所有剩余项目通过提供的测试。该测试是一个函数传递一个数组项和该数组内的项目的索引。只有当测试返回 `true`，该数组项将返回到结果数组中。

该过滤器的函数将被传递两个参数：当前数组项和它的索引。该过滤器函数必须返回 `true` 以包含在结果数组项中。注意，在筛选过程中原始数组不受影响。

【示例 12】 使用 `grep()` 函数筛选出大于等于 5 的数组元素，并返回一个新数组。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
    arr = jQuery.grep(arr, function(value, index){
        return value >= 5;
    });
    alert(arr);    //返回 5,6,7,8,9
})
</script>

```

```
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

反过来, 如果过滤掉大于等于 5 的数组元素, 则可设置第 3 个参数值为 true。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
    arr = jQuery.grep(arr, function(value, index){
        return value >= 5;
    },true);
    alert(arr);                //返回 1,2,3,4,0
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

jQuery 定义的 grep() 函数用法比较复杂, 但是使用 JavaScript 直接定义该函数的方法却很简单。详细代码如下:

```
<script type="text/javascript">
function grep( elems, callback, inv ) {                //模拟 jQuery 的 grep() 函数功能, 直接定义 grep() 函数
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ )    //遍历数组
        if ( !inv || !callback( elems[ i ], i ) )    //如果符合条件, 则存储该元素
            ret.push( elems[ i ] );
    return ret;                //返回筛选后的新数组
}
window.onload = function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
    arr = grep(arr, function(value, index){
        return value >= 5;
    }, true);
    alert(arr);                //返回 1,2,3,4,0
}
</script>
```

8.3.9 对数组进行转换

有时数据不一定是用户所需要的格式, 在以数据为中心的 Web 应用中另一个经常执行的操作就是把一组数值转换为另一组数值。尽管编写一个 for 循环从一个数组创建另一个数组是很简单的事情, 但是 jQuery 还是提供了一个比较实用的工具函数 map(), 它使类似的操作变得更加简单。

jQuery.map()函数拥有 grep()函数的过滤功能,同时还可以把当前数组根据处理函数处理后,映射为新的数组,甚至可以在映射过程中放大数组。该函数的具体用法如下:

```
jQuery.map(array, callback(elementOfArray, indexInArray))
jQuery.map(arrayOrObject, callback( value, indexOrKey))
```

- ☑ 参数 array 表示待转换的数组。
- ☑ 参数 arrayOrObject 表示待转换数组或对象。
- ☑ 参数 callback(elementOfArray, indexInArray)被每个数组元素调用,而且会给这个转换函数传递一个表示被转换的元素作为参数。函数可返回任何值。this 将是全局的 window 对象。

\$.map()方法是用于将数组或对象的每个项目新阵列映射到一个新数组的函数。在jQuery 1.6之前,\$.map()只支持遍历数组和类似数组的对象,jQuery 1.6 也支持遍历对象。类似数组的对象,如jQuery的collections,被当作数组。换句话说,如果一个对象有一个length属性和一个值在length-1上,那么它是一个遍历数组。

这个转换功能,是提供给调用此方法中的每个数组或对象的顶层元素的,并传递两个参数,该元素的值和其索引在数组或对象的key。

map()函数的用法与 grep()函数基本相似,包含两个参数:第一个参数表示被映射的数组,第二个参数表示数组元素处理转换函数。

作为第2个参数的转换函数会被每个数组元素调用,而且会给这个转换函数传递一个表示被转换的元素作为第1参数,元素的序号作为第2个参数被传递给转换函数。转换函数可以返回转换后的值。

如果转换函数返回值为null,则表示删除数组中对应的项目。如果转换函数返回值为一个包含值的数组,则表示将扩展原来的数组。

【示例 13】 把数组中的元素放大一倍之后,映射到一个新的数组中。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
        return elem * 2;
    });
    alert(arr); //返回 2,4,6,8
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

【示例 14】 如果修改转换函数,设置放大之后小于5的元素值,则返回null,即过滤掉数组中1和2两个元素。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
```

```

$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
        return elem * 2 > 5? elem * 2 : null;
    });
    alert(arr); //返回 6,8
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

【示例 15】如果在转换函数中设置返回值为数组，则可以在映射数组中扩大数组的长度。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
        return [elem,elem * 2];
    });
    alert(arr); //返回 1,2, 2,4, 3,6, 4,8
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

实际上，jQuery.map()工具函数的实现比较简单。如果使用 JavaScript 直接定义 map()函数，则代码如下：

```

<script type="text/javascript">
function map( elems, callback ) { //模仿 jQuery 的 map()函数功能，映射数组
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ ) { //遍历数组
        var value = callback( elems[ i ], i ); //执行回调转换函数，并获取函数返回值
        if ( value != null ) //如果返回值不为 null，则把该元素存储到临时数组中
            ret[ ret.length ] = value;
    }
    return ret.concat.apply( [], ret );
    //把临时数组连接到一个返回的空数组中，从而实现把数组类型的元素分开为元素，存储到返回数组中
}
</script>

```

8.3.10 把多个数组合并在一起

jQuery 定义了一个合并数组的函数 merge()，该函数能够把两个参数数组合并为一个新数组并返回。具

体用法如下:

```
jQuery.merge(first, second)
```

☑ 参数 first 表示第 1 个用来合并的数组, 元素是第 2 个数组加进来的。

☑ 参数 second 表示第 2 个数组合并到第 1 个, 保持不变。

\$.merge()操作形成一个数组, 其中包含两个数组的所有元素。从第 2 个追加的数组元素顺序将保存。

\$.merge()函数是破坏性的, 它改变了从第 2 个添加项目到第 1 个参数。

如果需要原始的第 1 个数组, 用户应该在调用\$.merge()前复制参数数组。不过 \$.merge()本身也可以用于此副本:

```
var newArray = $.merge([], oldArray);
```

此快捷方式将创建一个新的空数组合, 该数组合并了 oldArray 的内容, 有效地复制了数组。

在 jQuery 1.4 之前, 该参数应该是原生的 JavaScript 数组对象。如果情况并非如此, 参数不是原生的 JavaScript 数组对象, 应该使用\$.makeArray()工具函数把它转换为数组。

【示例 16】调用 merge()函数把数组 arr1 和 arr2 合并在一起, 并把合并后的数组传递给 arr1, 同时返回合并后的新数组。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var arr1 = [1,2,3,["a", "b", "c"]];
    var arr2 = [4,5,6,[7,8,9]];
    arr3 = jQuery.merge(arr1, arr2);
    alert(arr1); //返回数组[1,2,3,["a", "b", "c"],4,5,6,[7,8,9]]
    alert(arr1.length); //返回 8
    alert(arr3); //返回数组[1,2,3,["a", "b", "c"],4,5,6,[7,8,9]]
    alert(arr3.length); //返回 8
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>
```

如果直接使用 JavaScript 定义 merge()函数, 具体实现的代码如下:

```
<script type="text/javascript">
function merge( first, second ) { //模仿 jQuery 的 merge()函数功能, 合并数组
    var i = first.length, j = 0;
    if ( typeof second.length === "number" ) { //如果第 2 个参数为数组
        for ( var l = second.length; j < l; j++ ) { //遍历参数数组 2
            first[ i++ ] = second[ j ]; //逐一把数组 2 中的元素添加到参数数组 1 中
        }
    } else { //如果第二个参数为类数组的对象
        while ( second[j] !== undefined ) { //遍历该对象
            first[ i++ ] = second[ j++ ]; //逐一把对象中的成员添加到参数数组 1 中
        }
    }
}
```



```

first.length = i;           //重设数组 1 的 length 属性值
return first;               //返回合并后的数组
}
</script>

```

8.3.11 删除数组中重复元素

在 DOM 操作中,如果合并两个 jQuery 对象,可能会存在重复的 DOM 元素对象。为此, jQuery 专门定义了 `unique()` 函数,该函数可以把重复的 DOM 元素删除掉。考虑到 JavaScript 数组中可能会存在相同数值的元素,因此 jQuery 把该函数的功能限制在只处理删除 DOM 元素数组,而不能处理字符串或者数字数组。具体用法如下:

```
jQuery.unique(array)
```

参数 array 表示 DOM 元素的数组。

`unique()` 函数通过搜索的数组对象排序数组,并移除任何重复的节点。该功能只适用于普通的 JavaScript DOM 元素的数组,主要在 jQuery 内部使用。在 jQuery 1.4 中结果将始终按文档顺序返回。

【示例 17】 变量 `arr1` 存储了 3 个 DOM 元素,而 `arr2` 存储了两个 DOM 元素。合并之后,其中两个 DOM 元素是重复的。调用 `unique()` 函数之后,则删除这两个重复的选项,从而使合并后的数组中仅包含 3 个 DOM 对象。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $arr1 = $("#u1 li");
    var $arr2 = $(".red");
    var $arr3 = jQuery.merge($arr1, $arr2);
    var $arr4 = jQuery.unique($arr3);
    alert($arr1.length);           //返回 3
    alert($arr3.length);           //返回 3
    alert($arr4.length);           //返回 3
})
</script>
<title>上机练习</title>
</head>
<body>
<ul id="u1">
    <li>1</li>
    <li>2</li>
    <li class="red">3</li>
</ul>
<ul id="u2">
    <li class="red">4</li>
    <li>5</li>
    <li>6</li>
</ul>
</body>
</html>

```

8.3.12 在数组中查找指定值

jQuery 定义了 `inArray()` 工具函数用来查找数组中是否包含指定的值。具体用法如下：

`jQuery.inArray(value, array)`

☑ 参数 `value` 表示要搜索的值。

☑ 参数 `array` 表示一个数组，通过它来搜索。

该函数搜索数组中指定值，并返回它的索引，如果没有找到则返回 -1。

`inArray()` 函数类似于 JavaScript 的原生 `indexOf()` 方法，没有找到匹配元素时它返回 -1。如果数组第 1 个元素匹配 `value`，则返回 0。因为 JavaScript 将 0 视为 `false`，即 `0 == false`，但是 `0 != false`。如果检查在 `array` 中存在 `value`，只需要检查它是否不等于（或大于）-1 即可。

【示例 18】 定义一个数组，该数组包含 4 个元素，然后使用 `inArray()` 函数分别检查指定的值是否存在，以及如果存在，则给出在数组中的下标位置。演示效果如图 8.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    var arr = [ 4, "Pete", 8, "John" ];
    $("span:eq(0)").text(jQuery.inArray("John", arr));
    $("span:eq(1)").text(jQuery.inArray(4, arr));
    $("span:eq(2)").text(jQuery.inArray("Karl", arr));
})
</script>
<title>上机练习</title>
</head>
<body>
<h2>arr = [ 4, "Pete", 8, "John" ]</h2>
<div>"John"位于<span></span></div>
<div>4 位于<span></span></div>
<div>"Karl"位于<span></span>, 所以不存在</div>
</body>
</html>
```

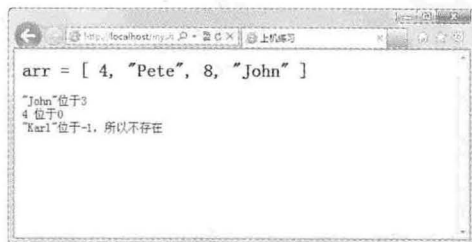


图 8.9 `inArray()` 函数的应用

8.4 缓 存

jQuery 在 1.2.3 版本中开始加入缓存功能，数据缓存的作用就是在一个元素上存取数据而避免循环引用

的风险。jQuery 通过 data()函数实现数据缓存的机制。

8.4.1 定义缓存

使用 data()工具函数可以为 jQuery 对象定义缓存数据。这些缓存数据被存放在匹配的 DOM 元素集中的所有 DOM 元素，同时返回保留缓存数据 value 的 jQuery。具体用法如下：

jQuery.data(element, key, value)

- ☑ 参数 element 表示要关联数据的 DOM 对象。
- ☑ 参数 key 表示存储的数据名。
- ☑ 参数 value 表示新数据值。

jQuery.data()方法允许在 DOM 元素上附加任意类型的数据，避免了循环引用的内存泄漏风险。注意，该工具目前并不提供在 XML 文档上跨平台设置，且 IE 浏览器不允许通过自定义属性附加数据。

【示例 19】 分别为导航列表中的 li 元素定义缓存数据，即列表选项的类型为 menu，同时为新闻列表中的 li 元素定义缓存数据，即列表选项的类型为 news。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
})
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<ul id="news">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
</body>
</html>
```

如果 jQuery 集合指向多个元素，则将为所有元素定义缓存数据。该函数在 DOM 元素上存放任何格式的数据，而不仅仅是字符串。

jQuery 数据缓存的实现方法其实很简单，如果打开 jQuery 技术框架源代码，搜索 “data: function(elem, name, data)” 关键词，可以找到下面一段代码：

```
var expando = "jQuery" + now(), uuid = 0, windowData = {};
jQuery.extend({
    cache: {},
```



```

data: function( elem, name, data ) {
    elem = elem == window ?           //如果元素为 window 对象，则设置 elem 为 windowData 对象
        windowData :
        elem;
    var id = elem[ expando ];           //为当前元素定义一个数据属性，并传递给 id 变量
    if ( !id )                          //如果不存在 id，则重赋一个动态值
        id = elem[ expando ] = ++uuid;
    //如果缓存数据对象中未存在特定数据的属性，则重设该属性
    if ( name && !jQuery.cache[ id ] )
        jQuery.cache[ id ] = {};
    //如果存在数据值参数，则把它存储到数据缓存对象
    if ( data !== undefined )
        jQuery.cache[ id ][ name ] = data;
    //如果存在属性名，则返回数据缓存对象中的对应属性，否则返回 id 值
    return name ?
        jQuery.cache[ id ][ name ] :
        id;
},

```

jQuery 首先声明 cache 缓存对象，初始化该对象为空。实际上，它的结构如下：

```

cache = {
    "uuid1":{
        "name1":value1,
        "name2":value2
    },
    "uuid2":{
        "name1":value1,
        "name2":value2
    }
}

```

每个 uuid 对应一个 elem 缓存数据，每个缓存对象可以由多个 name/value 对组成，而 value 可以是任何数据类型的。例如，可以为 elem 存一个如下的 JSON 数据片段：

```

$(elem).data('JSON',{
    "name":"张三",
    "age":12
})

```

expando 作为 elem 的一个新加属性，是为了防止与用户自定义的变量产生冲突，这里采用可变后缀的方式，从而最大限度避开名字冲突。

实际上，jQuery 通过一个公共数据缓存对象来存储不同 DOM 元素的数据，并通过一定方法区分不同元素的数据。为了方便元素访问自身的数据，则利用该元素及其一定的算法得到一个复合属性名，并利用这个复合属性名从公共缓存对象中读取自己的数据。也可以直接通过 JavaScript 方式定义一个 data() 函数，然后根据传进的元素对象，把特定名/值对数据添加到 cache 对象中。读写时，通过一定的算法区分不同元素的缓存数据，避免重复引用和迭代操作所造成的系统资源紧张。

```

<script type="text/javascript">
var expando = "jQuery" + now(),           //获取随机扩展数
uuid = 0,                                //uuid 起始值
windowData = {};                          //特殊顶级对象
cache = {},                              //公共缓存对象
function data( elem, name, data ) {       //DOM 元素的缓存数据读写函数
    elem = elem == window ?
        windowData :

```

```

    elem;
    var id = elem[ expando ];
    if ( !id )
        id = elem[ expando ] = ++uuid;
    if ( name && !cache[ id ] )
        cache[ id ] = {};
    if ( data !== undefined )
        cache[ id ][ name ] = data;
    return name ?
        cache[ id ][ name ] :
        id;
}
function now(){                                //以当前时间为基础获取一个随机数
    return +new Date;
}
</script>

```

为了方便 jQuery 对象操作, jQuery 又把全局函数 data() 绑定到 jQuery.fn 原型对象中, 从而实现在 jQuery 对象上直接调用 data() 方法。绑定代码如下:

```

jQuery.fn.extend({
    data: function( key, value ){
        var parts = key.split(".");
        parts[1] = parts[1] ? "." + parts[1] : "";
        if ( value === undefined ) {
            var data = this.triggerHandler("getData" + parts[1] + "!", [parts[0]]);
            if ( data === undefined && this.length )
                data = jQuery.data( this[0], key );
            return data === undefined && parts[1] ?
                this.data( parts[0] ) :
                data;
        } else
            return this.trigger("setData" + parts[1] + "!", [parts[0], value]).each(function(){
                jQuery.data( this, key, value );
            });
    },

```

8.4.2 读取缓存

data() 工具函数不仅可以定义缓存数据, 也可以读取缓存数据。具体用法如下:

```

jQuery.data(element, key)
jQuery.data(element)

```

- ☑ 参数 element 表示要关联数据的 DOM 对象。
- ☑ 参数 key 表示存储的数据名。

当传递 element 和 key 参数值时, 将返回具体的数据。如果直接传递一个 element 参数值, 则返回 Object。如果没有设置任何值, 那么将返回 null。如果 jQuery 集合指向多个元素, 则将只返回第 1 个元素的对应缓存数据。

调用没有参数的 jQuery.data(element) 时将获取一个作为 JavaScript 对象的所有值。jQuery 内部自身使用这个方法绑定数据, 如事件处理器, 所以不要以为它只包含数据存储的代码。

【示例 20】 针对上面示例, 可以分别获取 li 元素列表中的数据, 并根据 type 缓存数据的值, 分别显示不同的信息。演示效果如图 8.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
    $("li").each(function(index){
        if($(this).data("type") == "menu"){
            $(this).text("导航" + (index + 1))
        }
        else if($(this).data("type") == "news"){
            $(this).text("新闻" + (index + 1))
        }
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<ul id="news">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
</body>
</html>

```

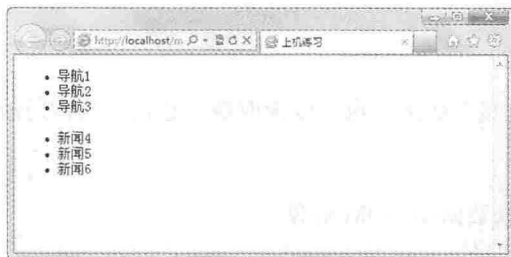


图 8.10 data()工具函数的应用

8.4.3 删除缓存

由于 jQuery 缓存对象是全局对象，因此在 Ajax 应用中，由于页面很少被刷新，缓存对象将会一直存在。随着调用 data() 函数操作次数增多，或者因使用不当，使得 cache 对象急剧膨胀，最终影响程序的性能。所以在使用 jQuery 数据缓存功能时，应及时清理缓存对象，jQuery 也提供了 removeData() 函数帮助用户手动

清除数据。

removeData()函数能够删除指定名称的缓存数据,并返回对应的jQuery对象。具体用法如下:

jQuery.removeData(element, [name])

- ☑ 参数 element 表示要移除数据的 DOM 对象。
- ☑ 参数 name 表示要移除的存储数据名。

jQuery.removeData()方法允许移除用 data()绑定的值。当带 name 参数调用时, jQuery.data()将删除那个特有的值;当不带任何参数时,所有的值将被移除。

【示例 21】 删除导航列表中 li 元素的 type 缓存数据,同时对 type 值为 news 的列表项,修改列表项包含的文本内容。演示效果如图 8.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
    $("li").each(function(index){
        if($(this).data("type") == "menu"){
            $(this).removeData("type");
        }
        else if($(this).data("type") == "news"){
            $(this).text("新闻" + (index + 1))
        }
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<ul id="menu">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<ul id="news">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
</body>
</html>
```

根据 jQuery 框架的运行机制,下面几种情况不需要手动清除数据缓存:

- ☑ 对 elem 执行 remove()操作, jQuery 会自动清除对象可能存在的缓存。
- ☑ 对 elem 执行 empty()操作,如果当前 elem 子元素存在数据缓存, jQuery 也会清除子对象可能存在的缓存,因为 jQuery 的 empty()实现其实是循环调用 remove()删除子元素。
- ☑ jQuery 复制节点 clone()方法不会复制 data 缓存,也就是说 jQuery 不会在全局缓存对象中分配一个

新节点存放新复制的 elem 缓存。



图 8.11 removeData()工具函数的应用

jQuery 在 clone()方法中把可能存在的缓存指向的属性（即 elem 的 expando 属性）替换成空。如果直接复制这个属性，就会导致原 elem 和新复制的 elem 都指向一个数据缓存，中间的操作都将会影响到两个 elem 的缓存变量。

把数据缓存一起复制有时候也是很有用的。例如，在拖动操作中，当单击源目标，elem 节点就会复制出一个半透明的 elem 副本开始拖动，并把 data 缓存复制到拖动层中。等到拖动结束，就可能取到当前拖动的 elem 相关信息。现在 jQuery 方法没有提供这样的处理，但在复制源目标的 data 时，把这些 data 都重新设置到复制出来的 elem 中，这样在执行 data(name, value)方法时，jQuery 会在全局缓存对象中开辟新空间。实现代码如下：

```
if (typeof($.data(currentElement)) == 'number') {  
    var elemData = $.cache[$.data(currentElement)];  
    for (var k in elemData) {  
        draggingDiv.data(k, elemData[k]);  
    }  
}
```

在上面代码中，\$.data(elem,name,data)包含 3 个参数，如果只有一个 elem 参数，这个方法返回它的缓存 key（即 uuid），利用这个 key 就可以得到整个缓存对象，然后把对象的数据都复制到新的对象中。

第 9 章

功能扩展

( 视频讲解：1 小时 32 分钟)

通过前面几章的系统学习，读者能够亲身体验到 jQuery 的强大，它所提供的有用命令及函数很庞大，并可以轻松地把这些工具捆绑在一起，以便更随心所欲地操作页面。即便如此，如果一些代码遵循一定的使用模式，并在开发中反复出现，那么我们应该有理由把这些代码添加到 jQuery 中，以便更加满意地完成日常开发工作。于是关于扩展的问题就成为 jQuery 框架最为重要、最核心的话题和实现。

扩展 jQuery 可以利用 jQuery 提供的现有代码基础。例如，通过创建新的 jQuery 命令（即包装器方法），可以自动继承 jQuery 强大的选择器机制的作用。如果能够在 jQuery 基础上进行二次开发，为什么还要从头开始浪费大量的时间和精力。因此，编写可重用的组件作为 jQuery 扩展是一个好习惯，也是聪明的工作方式。本章将详细讲解 jQuery 扩展实现方法，以及 jQuery 插件设计准则、模式、封装和优化方法。

9.1 自定义插件

jQuery 允许开发人员自定义 jQuery 扩展功能，并提供了友好的接口。jQuery 的易扩展性吸引越来越多的开发者和业余爱好者去研究、设计和使用 jQuery 插件。目前，全球有超过上千种不同应用需要的插件。使用这些插件能够帮助开发人员提升开发速度，节约劳动成本。最权威的插件可以从 jQuery 官方网站获取 (<http://plugins.jquery.com/>)。

9.1.1 jQuery 插件形式

jQuery 插件的开发包括 3 种：一种是类级别的插件开发，即给 jQuery 添加新的全局函数，相当于给 jQuery 类本身添加方法，jQuery 的全局函数就是属于 jQuery 命名空间的函数；另一种是对象级别的插件开发，即给 jQuery 对象添加方法；还有一种特殊形式就是扩展选择器。

1. 以 jQuery 方法的形式进行扩展

这种形式的插件是把一些常用或者重复使用的功能定义为函数，然后绑定到 jQuery 对象上，成为 jQuery 对象的一个扩展方法，对 jQuery 包装集进行操作的方法，即 jQuery 命令。

大部分 jQuery 插件都是这种形式的插件，这种插件是将对象方法封装起来，通过 jQuery 选择器获取 jQuery 对象进行操作，从而发挥 jQuery 强大的选择器优势。有很多 jQuery 内部方法，也是在 jQuery 脚本内部通过这种形式插入到 jQuery 框架中，如 `parent()`、`appendTo()`、`addClass()` 等方法。

2. 以工具函数的形式进行扩展

可以在\$(jQuery 的别名)上直接定义实用工具函数,把自定义的工具函数独立附加到jQuery命名空间下,作为jQuery作用域下一个公共函数被使用。例如,jQuery的ajax()方法就是利用这种途径内部定义的全局函数。由于全局函数没有绑定到jQuery对象上,故不能够在选择器获取的jQuery对象上直接调用,需要通过jQuery.fn()或者\$.fn()方式引用。

3. 以选择器的形式进行扩展

jQuery提供了强大的选择器,当然用户也可以自定义选择器,以满足特定环境下选择元素的需要。

9.1.2 自定义jQuery插件基本规则

jQuery开发团队制定了jQuery插件通用规则,为用户创建一个通用而可信的环境。因此,建议读者在自定义插件之前阅读并遵守这些规则,确保自定义插件与其他代码融合。遵守这些规则非常重要,它不仅保证插件代码的统一性,还能增加插件的成功几率。

1. 命名规则

jquery.plugin_name.js

其中,plug-in_name表示插件的名称。在这个文件中,所有全局函数都应该包含在名为plug-in_name的对象中。除非插件只有一个函数,则可以考虑使用jQuery.plugin_name()形式。

插件中的对象方法可以灵活命名,但是应保持相同的命名风格。如果定义多个方法,建议在方法名前添加插件名前缀,以保持清晰。不建议使用过于简短的名称、语义含糊的缩写名或公共方法名,如set()、get()等,这样很容易与外界的方法混淆。

2. 编码规则

编码规则如下:

- ☒ 所有新方法都附加到jQuery.fn对象上。
- ☒ 所有新功能都附加到jQuery对象上。

3. this 指针

插件内的this应该引用jQuery对象。

让所有插件在引用this时,知道从jQuery接收到哪个对象。所有jQuery方法都是在一个jQuery对象的环境中调用的,因此函数体中this关键字总是指向该函数的上下文,即this此时是一个包含多个DOM元素的数组。

4. 迭代元素

使用this.each()迭代匹配的元素。

插件应该调用this.each()来迭代所有匹配的元素,然后依次操作每个DOM元素。在this.each()方法体内,this就不再引用jQuery对象,而是引用当前匹配的DOM元素对象。

5. 勿忘返回值

插件应该有返回值,除了特定需求外,所有方法都必须返回jQuery对象。

一般都应该返回当前上下文环境中的jQuery对象,即this关键字引用的数组。通过这种方式,可以保持jQuery框架内方法的连续行为,即链式语法。如果破坏这种规则,就会给用户开发带来诸多不便。

如果匹配的对象集合被修改,则应该通过调用pushStack()方法创建新的jQuery对象,并返回这个新对象。如果返回值不是jQuery对象,则应该明确说明。

6. 语法严谨

插件中定义的所有方法或函数，在末尾都必须加上分号 (;)，以方便代码压缩。

7. 区别 jQuery 和 \$

在插件代码中总是使用 jQuery，而不是 \$。

\$ 并不总是等于 jQuery，这个很重要。如果用户使用 `var JQ = jQuery.noConflict()` 函数更改 jQuery 别名，那么就会引发错误。另外，其他 JavaScript 框架也可能使用 \$ 别名。

在复杂的插件中，如果全部使用 jQuery 代替 \$，又会让人难以接受这种复杂的写法。为了解决这个问题，建议使用如下插件模式：

```
(function($){
    //在插件包中使用$代替 jQuery
})(jQuery);
```

这个包装函数接收一个参数，该参数传递的是 jQuery 全局对象，由于参数被命名为 \$，因此在函数体内就可以安全使用 \$ 别名，而不用担心命名冲突。

9.1.3 使用 extend() 函数

为了方便用户创建插件，jQuery 自定义了 `jQuery.extend()` 和 `jQuery.fn.extend()` 方法。其中，`jQuery.extend()` 方法能够创建工具函数或者选择器，而 `jQuery.fn.extend()` 方法能够创建 jQuery 对象命令。

实际上，`extend()` 方法还可以实现更多功能，如把两个或更多的对象的内容汇集集成到第 1 个对象。具体用法如下：

```
jQuery.extend(target, [object1], [objectN])
jQuery.extend([deep], target, object1, [objectN])
```

- ☑ 参数 `target` 表示一个对象，如果附加的对象被传递给这个方法，那么它将接收新的属性。如果它是唯一的参数，则将扩展 jQuery 的命名空间。
- ☑ 参数 `object1` 表示一个对象，它包含额外的属性合并到第 1 个参数。
- ☑ 参数 `objectN` 表示包含额外的属性合并到第 1 个参数。
- ☑ 参数 `deep` 如果是 `true`，表示合并成为递归，又叫做深拷贝。

如果为 `extend()` 方法提供两个或多个对象，则对象的所有属性都被添加到目标对象上。

如果只有一个参数提供给 `extend()`，意味着目标参数被省略。在这种情况下，jQuery 对象本身被默认为目标。这样，就可以在 jQuery 的命名空间下添加新的功能。这可用于添加新的方法到 jQuery。

注意，目标对象（第 1 个参数）将被修改，也将通过 `extend()` 方法返回。然而，如果想保留原对象，可以通过传递一个空对象作为目标：

```
var object = $.extend({}, object1, object2);
```

通过 `extend()` 方法合并执行时，默认不是递归操作。如果第 1 个对象的属性本身是一个对象或数组，则将完全用与第 2 个对象相同的 key 重写一个属性，这些值不合并。然而，`true` 为第 1 个函数参数，对象将被递归合并。未定义的属性不会被复制，但是对象原型的继承属性将被复制。

【示例 1】 使用 `extend()` 工具函数把对象 `object2` 合并到 `object1` 中，此时 `object1` 就包含了 4 个属性，其中 `banana` 属性值被覆盖。最后返回的结果是 `object1 = {apple: 0, banana: {price: 200}, cherry: 97, durian: 100}`。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
```

```

<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var object1 = {
        apple: 0,
        banana: {weight: 52, price: 100},
        cherry: 97
    };
    var object2 = {
        banana: {price: 200},
        durian: 100
    };
    $.extend(object1, object2);
    alert(object1.banana.weight);      //返回 undefined
    alert(object1.banana.price);      //返回 200
})
</script>
<style type="text/css">
</style>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

如果不希望 object2.banana 覆盖 object1.banana，则可以添加一个 true 参数。具体代码如下：

```
$.extend(true, object1, object2);
```

则返回的对象为 object1 = {apple: 0, banana: {weight: 52, price: 200}, cherry: 97, lime: 100}。

【示例 2】 利用 extend() 工具函数也可以修改或者设置默认值。在下面示例中将修改 settings 对象的属性值，使 settings.validate 等于 true，settings.name 等于 bar，最后返回 settings = { validate: true, limit: 5, name: "bar" }。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var settings = { validate: false, limit: 5, name: "foo" };
    var options = { validate: true, name: "bar" };
    jQuery.extend(settings, options);
})
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

下面代码为参数对象设置默认值，如果用户传递了新值，则使用用户传递的值，否则使用默认值，最后返回：


```
settings = { validate: true, limit: 5, name: "bar" }
empty = { validate: true, limit: 5, name: "bar" }
```

【示例 3】在 jQuery 命名空间上创建两个公共函数，然后就可以在页面中调用这两个公共函数。当单击按钮后，浏览器会弹出提示对话框，要求输入两个值，并提示两个值的大小。演示效果如图 9.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.extend({ //扩展 jQuery 的公共函数
    minValue : function(a,b){ //比较两个参数值，返回最小值
        return a<b?a:b;
    },
    maxValue : function(a,b){ //比较两个参数值，返回最大值
        return a<b?b:a;
    }
})
$(function(){
    $("input").click(function(){
        var a = prompt("请输入一个数值? ");
        var b = prompt("请再输入一个数值? ");
        var c = jQuery.minValue(a,b);
        var d = jQuery.maxValue(a,b);
        alert("你输入的最大值是: " + d + "\n 你输入的最小值是: " + c);
    });
});
</script>
<style type="text/css">
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="测试自定义 jQuery 函数" />
</body>
</html>
```



图 9.1 自定义 jQuery 工具函数

在实际开发中，经常使用 jQuery.extend() 方法为插件方法传递系列选项结构的参数。例如：

```
function fn(options){
    var options = jQuery.extend({ //默认参数选项列表
```

```

        name1 : value1,
        name2 : value2,
        name3 : value3
    }, options);
    //函数体
}

```

//使用函数的参数覆盖或合并到默认参数选项列表中

这样当在调用该方法时，如果传递新的参数值，就会覆盖掉默认的参数选项值，或者向函数参数添加新的属性和值；如果没有传递参数，则保持并使用默认值。例如，在下面几个函数调用中，分别传入新值、添加新参数、保持默认值。

```

fn({name1 : value2, name2 : value3, name3 : value1}); //覆盖新值
fn({name4 : value4, name5 : value5 });               //添加新选项
fn();                                                  //保持默认参数值

```

jQuery.extend()方法的对象合并机制比传统的逐个检测参数灵活且简洁，使用命名参数添加新选项也不会影响已编写的代码风格，让代码变得更加直观明白。

jQuery.extend()方法能够创建全局函数，而 jQuery.fn.extend()方法可以创建 jQuery 对象方法。jQuery.fn.extend()方法仅包含一个参数，该参数是一个对象直接量，以名/值对形式组成多个属性，名称表示方法名称，值表示函数体。因此，在这个对象直接量中可以附加多个属性，为 jQuery 对象同时定义多个方法。

【示例 4】为 jQuery 自定义一个 test()的方法，利用这个方法获取当前 jQuery 对象中包含的每个元素的节点名称，然后就可以在 jQuery 选择器中直接调用 test()了。演示效果如图 9.2 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
jQuery.fn.extend({
    test : function(){
        return this.each(function(){
            alert(this.nodeName);
        });
    }
})
$(function(){
    $("body *").click(function(){
        $(this).test();
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
</body>
</html>

```

//自定义 jQuery 命令

//调用 jQuery 对象方法



图 9.2 自定义 jQuery 命令

9.1.4 自定义 jQuery 函数

jQuery 内置的很多方法都是通过全局函数实现的。所谓全局函数，就是 jQuery 对象的方法，实际上就是位于 jQuery 命名空间内部的函数，即工具函数。这些函数有一个共同特征，就是不直接操作 DOM 元素，而是用这些函数来操作 JavaScript 非元素对象，或者执行其他非对象的特定操作，如 jQuery 的 each() 函数和 noConflict() 函数。

这类形式扩展也称为类级别开发，类级别的插件开发最直接的理解就是给 jQuery 类添加类方法，可以理解为添加静态方法。关于类级别的插件开发可以采用如下几种形式进行扩展。

☒ 添加一个新的全局函数

添加一个全局函数，只需如下定义：

```
jQuery.foo = function() {
    alert('This is a test. This is only a test.');
```

☒ 添加多个全局函数

添加多个全局函数，可以采用如下形式定义：

```
jQuery.foo = function() {
    alert('This is a test. This is only a test.');
```

```
};
```

```
jQuery.bar = function(param) {
    alert('This function takes a parameter, which is "' + param + '"');
```

调用时与调用函数一样，都为 “jQuery.foo();jQuery.bar();” 或者 “\$.foo();\$.bar("bar");”。

☒ 使用 jQuery.extend() 函数

```
jQuery.extend({
    foo: function() {
        alert('This is a test. This is only a test.');
```

☒ 使用命名空间

虽然在 jQuery 命名空间中，禁止使用大量的 JavaScript 函数名和变量名，但是仍然不可避免某些函数或变量名将与其他 jQuery 插件冲突，因此习惯将一些方法封装到另一个自定义的命名空间。

```
jQuery.myPlugin = {
    foo:function() {
        alert('This is a test. This is only a test.');
```



```

bar:function(param) {
    alert('This function takes a parameter, which is "' + param + '".');
}
};

```

采用命名空间的函数仍然是全局函数，调用时采用的方法如下：

```

$.myPlugin.foo();
$.myPlugin.bar('baz');

```

通过使用独立的插件名这个技巧，可以避免命名空间内函数的冲突。

在 9.1.3 节中曾经介绍了使用 `jQuery.extend()` 方法可以扩展 jQuery 工具函数。当然，也可以使用下面方法快速定义 jQuery 工具函数。

【示例 5】 在 jQuery 命名空间上创建两个公共函数。也就是说，如果要向 jQuery 命名空间上添加一个函数，只需要将这个新函数指定为 jQuery 对象的一个属性即可，其中 jQuery 对象名也可以简写为 \$。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.minValue = function(a,b){
    return a<b?a:b;
};
jQuery.maxValue = function(a,b){
    return a<b?b:a;
}
</script>
<title>上机练习</title>
</head>
<body>
</body>
</html>

```

考虑到 jQuery 的插件越来越多，因此在使用时可能会遇到自己的插件名与第三方插件名发生冲突的问题。为了避免这个问题，建议把属于自己的插件都封装在一个对象中。

【示例 6】 针对上面两个创建的工具局函数，可以把它们封装在自定义对象中。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.MyExtend = {
    minValue : function(a,b){
        return a<b?a:b;
    },
    maxValue : function(a,b){
        return a<b?b:a;
    }
}
</script>

```

```

<title>上机练习</title>
</head>
<body>
</body>
</html>

```

尽管把这些函数当成全局函数看待，但是从技术层面分析，它们现在都是全局 jQuery 函数的方法，因此在调用这些函数时，应该使用下面方式：

```

$(function(){
    var c = jQuery.MyExtend.minValue(a,b);
    var d = jQuery.MyExtend.maxValue(a,b);
})

```

注意，考虑到安全性不建议以一种简写的方式（即使用\$代替jQuery）书写，应该在编写的插件中始终使用 jQuery 来调用 jQuery 方法。

9.1.5 自定义 jQuery 命令

除了工具函数外，jQuery 中的大多数功能都是通过 jQuery 对象的方法提供的，这些对象方法对于 DOM 操作来说非常方便。自定义 jQuery 函数通过为 jQuery 对象添加属性即可，而自定义 jQuery 命令可以通过为 jQuery.fn 对象添加属性。实际上，jQuery.fn 对象就是 jQuery.prototype 原型对象的别名，使用别名更方便引用。

对象级别的插件开发需要如下两种形式：

☑ 形式 1

```

(function($){
    $.fn.extend({
        pluginName:function(opt,callback){
            // Our plugin implementation code goes here.
        }
    })
})(jQuery);

```

☑ 形式 2

```

(function($) {
    $.fn.pluginName = function() {
        // Our plugin implementation code goes here.
    };
})(jQuery);

```

上面定义了一个 jQuery 函数，形参是\$，函数定义完成之后，把 jQuery 这个实参传递进去，立即调用执行。这样的好处是在写 jQuery 插件时，可以使用\$这个别名，而不会与 prototype 等其他第三方插件或者外部脚本中的标识符发生冲突。

【示例 7】 如果单击页面中的测试按钮，即可弹出一个提示对话框，提示“自定义 jQuery 命令成功”的信息。演示效果如图 9.3 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.fn.test = function(){ //自定义 jQuery 命令
    alert("自定义 jQuery 命令成功");
}

```

```

}
$(function(){
    $("input").click(function(){           //绑定 click 事件
        $(this).test();                   //在当前的 jQuery 对象上调用 test()方法
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="jQuery 测试" />
</body>
</html>

```



图 9.3 自定义 jQuery 命令

注意, 在使用 jQuery 命令时, 方法的函数体内的 `this` 关键字总是引用当前 jQuery 对象, 因此可以对上面的方法进行重写, 实现动态提示信息。这样当单击按钮时, 就会提示当前元素的节点名称。演示效果如图 9.4 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
jQuery.fn.test = function(){
    alert(this[0].nodeName);           //提示当前 jQuery 对象的 DOM 节点名称
}
$(function(){
    $("input").click(function(){       //绑定 click 事件
        $(this).test();               //在当前的 jQuery 对象上调用 test()方法
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="jQuery 测试" />
</body>
</html>

```



图 9.4 自定义 jQuery 命令

在上面示例中可以看到, 由于 jQuery 选择器返回的是一个数组类型的 DOM 节点集合, this 指针就指向当前这个集合, 故显示当前元素的节点名称, 必须在 this 后面指定当前元素的序号。

如果 jQuery 选择器匹配多个元素, 如何准确指定当前元素对象呢? 用户不妨在当前 jQuery 对象方法的环境中调用 each() 方法, 通过隐式迭代的方式, 让 this 指针依次引用每一个匹配的 DOM 元素对象, 这样也能够让插件与 jQuery 内置方法保持一致性。

【示例 8】针对示例 7 做进一步的修改, 调用 jQuery 命令 each 遍历当前 jQuery 对象中包含的所有 DOM 元素, 然后逐一读取其中每个元素的节点名称。演示效果如图 9.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.fn.test = function(){
    this.each(function(){ //遍历所有匹配的元素, 此处的 this 表示对象集合, 即 jQuery 对象
        alert(this.nodeName); //显示当前元素的节点名称, 此处的 this 表示元素对象
    });
}

$(function(){
    $("body *").click(function(){ //选择 body 元素下所有元素
        $(this).test(); //为当前元素调用 test(), 提示当前 DOM 元素对象的节点名称
    });
})
</script>
<style type="text/css">
</style>
<title>上机练习</title>
</head>
<body>
<input type="button" value="jQuery 测试" />
<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
</body>
</html>
```

jQuery 遵循链式语法规则, 为了让自定义的 jQuery 命令也符合这种规则, 还需要对上面的方法进行完善, 即在每个插件方法中返回一个 jQuery 对象, 方法需要明确返回值的情况除外。返回的 jQuery 对象通常

就是 `this` 所引用的对象。如果使用 `each()` 方法迭代 `this`，则可以直接返回迭代的结果。



图 9.5 动态显示节点名称

【示例 9】 针对示例 8 做进一步的修改，让自定义插件返回当前的 jQuery 对象，然后就可以在应用示例中连写行为了。在下面示例中，先弹出提示节点名称的信息，然后使用当前节点名称改写当前元素内包含的信息，最后再慢慢隐藏该元素。演示效果如图 9.6 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.fn.test = function(){
    return this.each(function() { //返回迭代的 jQuery 对象
        alert(this.nodeName);
    });
}
$(function(){
    $("body *").click(function(){
        $(this).test().html(this.nodeName).hide(4000); //链式语法
    });
})
</script>
<title>上机练习</title>
</head>
<body>
<input type="button" value="jQuery 测试" />
<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
</body>
</html>
```

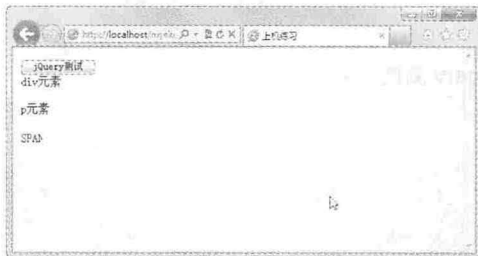


图 9.6 jQuery 方法链式语法应用

9.1.6 自定义选择器

jQuery 允许开发人员扩展选择器功能，用户可以根据个人开发需要创建个性化的选择器，以满足特殊选择操作。首先读者应该了解 jQuery 选择器的工作机制。

jQuery 选择器使用一组正则表达式来分析选择符。然后，针对所解析出来的每一个选择符执行一个函数，这个函数称为选择器函数。最后，根据这个选择器函数的返回值是否为 true，决定是否保留当前元素，这样就可以找到所要匹配的元素节点。

例如，针对下面这个基本选择器，可以选择所有匹配的 p 元素的前面 2 个元素。代码如下：

```
$("p:lt(2)")
```

当 jQuery 解析这个选择器时，首先找出当前范围内所有的 p 元素，然后隐式遍历这些 p 元素，并逐个将这些 p 元素作为参数，将参数 2 都传递给 lt() 函数。lt() 函数的代码如下：

```
lt: function(elem, i, match){
    return i < match[3] - 0;
},
```

- ☑ 第 1 个参数表示当前遍历的 DOM 元素对象。
- ☑ 第 2 个参数表示当前 DOM 元素对象在所有匹配元素中的索引位置，从 0 开始。
- ☑ 第 3 个参数表示正则表达式执行匹配后返回的数组对象，相当于 match() 方法返回的子表达式所匹配的信息。其正则表达式如下：

```
match: {
    POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^)]$)/
}
```

结合上面的示例代码，match 参数数组的元素组成说明如下。

- ☑ match[0]: 表示 “:lt(2)” 部分字符串。
- ☑ match[1]: 表示选择器引导符，即表示 “:” 字符。
- ☑ match[2]: 表示选择器函数，即表示 “lt” 字符串。
- ☑ match[3]: 表示选择器函数中的序号参数，即表示 1 字符，它非常有用，在编写选择器函数时将会用到。
- ☑ match[4]: 表示选择器函数中的特殊参数。在此没有体现，例如，p:lt(a(b)) 选择器，match[4] 就匹配 “(b)” 字符串部分。

明白了 jQuery 选择器的设计思路，就可以自定义选择器。

【示例 10】 jQuery 提供了 :gt 和 :eq 选择器，但是没有定义 :ge (大于等于) 和 :le (小于等于) 等选择器。下面就来自定义 :ge 和 :le 选择器。

首先，模仿上面的方法设计选择器函数，代码如下：

```
le: function(elem, i, match){
    return i < match[3] - 0 || i == match[3] - 0;
},
ge: function(elem, i, match){
    return i > match[3] - 0 || i == match[3] - 0;
},
```

在上面两个函数中，le() 通过比较参数 i 的值与匹配元素的序号 (match[3]) 的大小相等关系，决定返回 true 或者 false。通过 match[3] - 0 表达式，强制转换 match[3] 值为数值型数据，而 ge() 函数正好相反。

然后，把上面的选择器函数添加到 jQuery 选择器对象上即可。在 jQuery 框架中，jQuery.expr[":"] 表示 jQuery 选择器对象的别名，它等于 jQuery.expr.filters，也等于 Sizzle.selectors.filters。实现的代码如下：

```
jQuery.expr[":"].le = function(elem, i, match){    //自定义小于等于选择器
    return i < match[3] - 0 || i == match[3] - 0;
```

```

}
jQuery.expr[":"].ge = function(elem, i, match){    //自定义大于等于选择器
    return i > match[3] - 0 || i == match[3] - 0;
}

```

最后，尝试利用自定义选择器来选择元素并定义样式。在下面示例中，选择序号等于或者小于 2 的 p 元素，设置它们的字体颜色为红色；选择序号等于或者大于 2 的 p 元素，设置它们的背景颜色为浅灰色。演示效果如图 9.7 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.expr[":"].le = function(elem, i, match){
    return i < match[3] - 0 || i == match[3] - 0;
}
jQuery.expr[":"].ge = function(elem, i, match){
    return i > match[3] - 0 || i == match[3] - 0;
}
$(function(){
    $("p:le(2)").css("color","red");
    $("p:ge(2)").css("background","#ddd");
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
<p>段落文本 4</p>
<p>段落文本 5</p>
<p>段落文本 6</p>
</body>
</html>

```

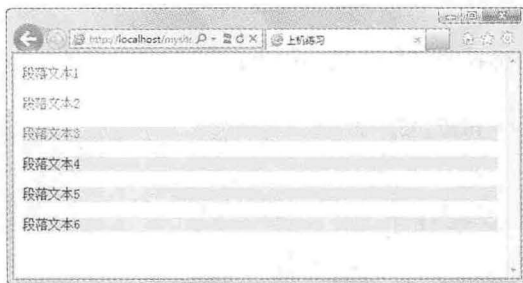


图 9.7 自定义选择器

另外，用户也可以使用 jQuery.extend() 方法来扩展 jQuery.expr[":"] 对象的方法。具体实现的代码如下：

```

jQuery.extend(jQuery.expr[":"],{
    le : function(elem, i, match){
        return i < match[3] - 0 || i == match[3] - 0;
    }
});

```

```

    },
    ge : function(elem, i, match){
        return i > match[3] - 0 || i == match[3] - 0;
    }
})

```

用户可以对 jQuery 默认选择器进行重写或者优化, 重写的方法是直接覆盖该方法。例如, 针对 `:nth-child` 选择器, 它能够匹配指定位置的元素, 或者选择奇偶元素。下面示例可以把所有匹配的 `li` 元素中第 2 元素设置为红色字体样式:

```

$(function(){
    $("li:nth-child(2)").css("color","red");
})

```

在 CSS 3 规范中, `:nth-child()` 伪类选择器的功能是非常强大的, 它不仅能够接收整数参数, 还可以接收 `an+b` 形式的任何表达式。如果某项位置等于这个表达式, 或者等于 `n` 在任何整数值下计算出的值, 这个项就匹配, 如 `3n+2` 表达式, 就会匹配 2、5、8 等位置上的元素。下面就根据这个计算原理优化 `:nth-child` 选择器的功能。

要覆盖 `nth-child()` 方法, 可以通过两种途径实现:

☑ 直接覆盖。结构代码如下:

```

jQuery.expr[":"].nth-child = function(elem, match){
    //函数体
}

```

但是, 考虑到 `nth-child` 字符串中的减号是运算符, 故建议直接使用第 2 种方法进行定义。

☑ 调用 `jQuery.extend()` 方法重写。结构代码如下:

```

jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){
        //函数体
    }
})

```

在 jQuery 框架代码中, 找到 `:nth-child()` 选择器的正则表达式直接量如下所示:

```

match: {
    CHILD: /(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
}

```

通过观察分析可以看到对于 `a(b(c))` 格式的伪类选择符, `match` 数组中的元素的含义如下。

- ☑ `match[0]`: 表示 “`a(b(c))`” 部分字符串。
- ☑ `match[1]`: 表示选择器引导符, 即表示 “`:`” 字符。
- ☑ `match[2]`: 表示选择器函数名, 即表示 “`a`” 字符串。
- ☑ `match[3]`: 表示选择器函数参数, 即表示 “`b(c)`” 字符串。
- ☑ `match[4]`: 表示选择器函数中的特殊参数, 即表示 “`(c)`” 字符串。

然后, 在 jQuery 框架代码中, 找到 `nth-child()` 函数体的代码如下:

```

CHILD: function(elem, match){
    var type = match[1], node = elem;
    switch (type) {
        case 'nth':
            var first = match[2], last = match[3];
            if ( first == 1 && last == 0 ) {
                return true;
            }
            var doneName = match[0],
                parent = elem.parentNode;

```



```

        if ( parent && (parent.sizcache !== doneName || !elem.nodeType) ) {
            var count = 0;
            for ( node = parent.firstChild; node; node = node.nextSibling ) {
                if ( node.nodeType === 1 ) {
                    node.nodeType = ++count;
                }
            }
            parent.sizcache = doneName;
        }
        var diff = elem.nodeType - last;
        if ( first == 0 ) {
            return diff == 0;
        } else {
            return ( diff % first == 0 && diff / first >= 0 );
        }
    }
},

```

根据上面函数体的设计思路，nth-child()函数需要重写如下代码：

```

jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){ //按 an+b 公式匹配元素，参数 elem 表示遍历元素，match 表示匹配返回
        的数组
        var index = $(elem).parent().children().index(elem) + 1; //重新计算序号，起始位置为 1，而不是 0
        var num = match[3].match(/((\d+)?n)?((\d+)?)/); //使用 match()方法匹配 a(b(c))格式中的“b(c)”字符串
        if(num[2] == undefined){ //如果 an+b 公式中，a 未知，则直接返回 b 作为序号
            return index == num[3];
        }
        else if(num[3] == undefined){ //如果 an+b 公式中，b 未知，则直接返回 0 作为序号
            num[3] = 0;
        }
        return (index-num[3]) % num[2] == 0; //最后根据公式测试当前元素是否匹配
    }
})

```

在上面这个函数中，首先从同辈中找到当前节点的索引，由于 CSS 3 选择器规定:nth-child()伪选择器的参数序号从 1 开始，故需要重新计算 index 序号值。

然后，根据 $an+b$ 公式反向推导， $an+b=y$ ，则 $n=(y-b)/a$ ，也就是说某个位置的元素，它的位置序号加 1 之后（即在上面公式中表示变量 y ）。如果参与公式计算之后，所得数值为一个整数，则说明该元素可以匹配。例如，针对 $3n+2$ 公式，当 n 等于 0 时，它可以匹配第 2 个元素；当 n 等于 1 时，它可以匹配第 5 个元素；当 n 等于 2 时，它可以匹配第 8 个元素，依此类推。

【示例 11】 覆盖了 jQuery 的 nth-child()选择器之后，就可以进行测试，本示例将匹配 $3n+2$ 公式中的所有元素，凡是匹配元素的字体颜色显示为红色。演示效果如图 9.8 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){
        var index = $(elem).parent().children().index(elem) + 1;

```

```

var num = match[3].match(/((\d+)\?)+(\d+)\?/);
if(num[2] == undefined){
    return index == num[3];
}
else if(num[3] == undefined){
    num[3] = 0;
}
return (index-num[3]) % num[2] == 0;
}
})
$(function(){
    $("p:nth-child(3n+2)").css("color","red");    //匹配 3n+2 位置上的元素
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
<p>段落文本 4</p>
<p>段落文本 5</p>
<p>段落文本 6</p>
</body>
</html>

```

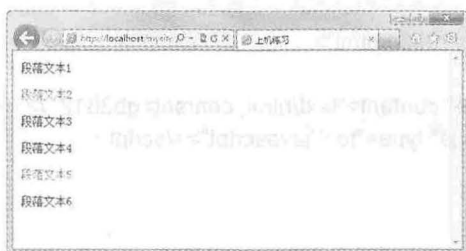


图 9.8 优化:nth-child()选择器

9.2 封装和优化插件

如果要对外发布自定义插件，用户应该对此插件进行封装和优化，封装插件应该符合规范，只有这样所创建的插件才能够被其他用户接受。

9.2.1 封装插件

封装 jQuery 插件的第 1 步是定义一个独立域，代码如下：

```

(function($){
    //自定义插件代码
})(jQuery)    //封装插件

```

确定创建插件类型，选择创建方式。例如，创建一个设置元素字体颜色的插件，则应该创建 jQuery 对象方法。考虑到 jQuery 提供了插件扩展方法 `extend()`，调用该方法定义插件会更为规范。代码如下：

```

(function($){
    $.extend($.fn,{
        //函数列表
    })
})(jQuery) //封装插件

```

一般插件都会接收参数，用来控制插件的行为，根据 jQuery 设计习惯，可以把所有参数以列表形式封装在选项对象中。例如，对于设置元素字体颜色的插件，应该允许用户设置字体颜色，同时还应考虑如果用户没有设置颜色，则应确保使用默认色进行设置。实现代码如下：

```

(function($){
    $.extend($.fn,{
        color : function(options){ //自定义插件名称
            var options = $.extend({ //参数选项对象处理
                bcolor : "white", //背景色默认值
                fcolor : "black" //前景色默认值
            },options);
            return this.each(function(){ //返回匹配的 jQuery 对象
                $(this).css("color", options.fcolor); //遍历设置每个 DOM 元素字体颜色
                $(this).css("backgroundColor", options.bcolor); //遍历设置每个 DOM 元素背景颜色
            })
        }
    })
})(jQuery); //封装插件

```

完成插件封装之后，应该测试自定义的 color() 方法。演示效果如图 9.9 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
(function($){
    $.extend($.fn,{
        color : function(options){
            var options = $.extend({
                bcolor : "white",
                fcolor : "black"
            },options);
            return this.each(function(){
                $(this).css("color", options.fcolor);
                $(this).css("backgroundColor", options.bcolor);
            })
        }
    })
})(jQuery); //封装插件
$(function(){
    $("p").color({
        bcolor : "blue",
        fcolor : "red"
    });
})
</script>

```

```

<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
<p>段落文本 4</p>
<p>段落文本 5</p>
<p>段落文本 6</p>
</body>
</html>

```

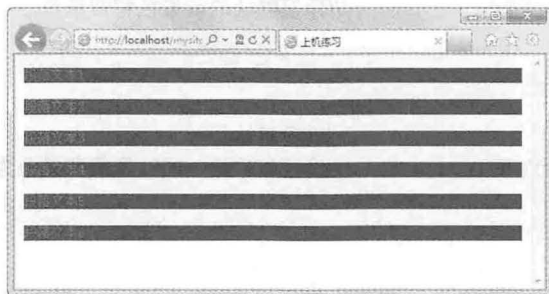


图 9.9 封装 jQuery 插件

9.2.2 优化插件

用户如果要发布自定义 jQuery 插件，应该确保插件的开放性和封闭性，能够方便其他用户使用，同时还应避免冲突或者破坏。

1. 允许定义默认设置

插件一般都需要用户设置参数，这个参数是灵活应用插件的体现，当然开发人员也应该考虑到用户不设置参数时该如何设置默认值，避免程序出错。同时，可以考虑开放插件的默认设置，这对于插件使用者，会更容易使用和修改。为了探讨这个话题，下面以示例进行讲解。

【示例 12】 继续以 9.2 节的代码为例进行说明，把其中的参数默认值作为 \$.fn.color 对象的属性单独设计，然后借助 jQuery.extend() 覆盖原来参数选项即可。在 color() 函数中，\$.extend() 方法能够使用参数 options 覆盖默认的 defaults 属性值。如果没有设置 options 参数值，则使用 defaults 属性值。由于 defaults 属性是单独定义的，可以在页面中预设前景色和背景色，然后就可以多次调用 color() 方法，完整的示例代码如下。演示效果如图 9.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
(function($){
    $.extend($.fn,{
        color : function(options){
            var options = $.extend({}, $.fn.color.defaults, options);    //覆盖原来参数
            return this.each(function(){

```



```

        $(this).css("color", options.fcolor);
        $(this).css("backgroundColor", options.bcolor);
    })
}
})
$.fn.color.defaults = {                                //独立设置$.fn.color 对象的默认参数值
    bcolor : "white",
    fcolor : "black"
}
})(jQuery);
$(function(){
    $.fn.color.defaults = {                                //预设默认的前景色和背景色
        bcolor : "red",
        fcolor : "blue"
    }
    $("p").color();                                       //设置默认色
    $("p:gt(2)").color({bcolor: "#fff"});               //设置默认色, 同时覆盖背景色为白色
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
<p>段落文本 4</p>
<p>段落文本 5</p>
<p>段落文本 6</p>
</body>
</html>

```

通过这种开发插件默认参数的做法, 用户不再需要重复定义参数, 可以节省开发时间。

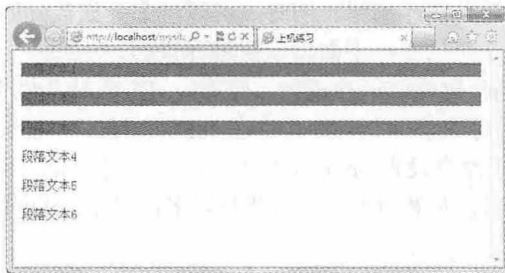


图 9.10 动态定义默认值

2. 让插件具有功能扩展性

实际上在封装插件时, 开发人员是无法确保把所有功能都封装进去, 也没有办法预知用户的所有需求, 此时最友好的方式就是让用户自己设计或者添加部分功能, 使该插件满足不同用户的不同需求。

【示例 13】 继续以 9.2.1 节的示例为基础, 为 color() 命令添加一个格式化的扩展功能, 这样用户在设置颜色的同时, 还可以根据需要进行格式化功能设计, 如加粗、斜体、放大等。

通过开放的方式定义了一个 format() 功能函数, 在这个功能函数中默认没有进行格式化设置, 然后在 color() 函数体内利用这个开放性功能函数格式化当前元素内的 HTML 字符串。最后调用 color() 命令, 在调用前分别扩展它的格式化功能, 演示效果如图 9.11 所示。

第9章 功能扩展

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
(function($){
    $.extend($.fn,{
        color : function(options){
            var options = $.extend({}, $.fn.color.defaults, options); //覆盖原来参数
            return this.each(function(){
                $(this).css("color", options.fcolor);
                $(this).css("backgroundColor", options.bcolor);
                var _html = $(this).html(); //获取当前元素包含的 HTML 字符串
                _html = $.fn.color.format(_html); //调用格式化功能函数对其进行格式化
                $(this).html(_html); //使用格式化的 HTML 字符串重写当前元素内容
            })
        }
    })
    $.fn.color.defaults = { //独立设置$.fn.color 对象的默认参数值
        bcolor : "white",
        fcolor : "black"
    }
    $.fn.color.format = function(str){ //开放的功能函数
        return str;
    };
})(jQuery);
$(function(){
    $.fn.color.defaults = { //预设默认的前景色和背景色
        bcolor : "#eea",
        fcolor : "red"
    }
    $.fn.color.format = function(str){ //扩展 color()插件的功能，使内部文本加粗显示
        return "<strong>" + str + "</strong>";
    }
    $("p").color();
    $("p:gt(2)").color({bcolor:"#fff"});
    $.fn.color.format = function(str){ //扩展 color()插件的功能，使内部文本放大显示
        return "<span style='font-size:30px;'>" + str + "</span>";
    }
    $("p:gt(4)").color();
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
<p>段落文本 4</p>
<p>段落文本 5</p>

```



```
<p>段落文本 6</p>
</body>
</html>
```

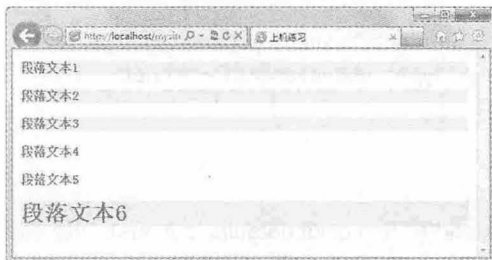


图 9.11 开放 color() 插件

允许用户自定义功能设置，以覆盖插件默认的功能，从而方便其他用户以当前插件为基础进一步扩写插件。

3. 让插件具有隐私功能

在设计插件时必须考虑插件暴露和隐私的关系，该暴露的功能就应该完全开放，不该暴露的属性或者成员就应该保护它的隐私性。一旦被暴露，则任何对于参数或者语义的改动也许会破坏向后的兼容性。如果不能确定应该暴露特定的函数，那么就on必须考虑如何进行保护的问题。

当插件包含很多函数时，在设计时希望这么多函数不搅乱命名空间，也不会被完全暴露，唯一的方法就是使用闭包。为了创建闭包，可以将整个插件封装在一个函数中。

【示例 14】 继续以 9.2.1 节示例进行讲解，为了验证用户在调用 color() 方法时所传递的参数合法，不妨在插件中定义一个参数验证函数，但是该验证函数是不允许外界侵入或者访问的，此时可以借助闭包把它隐藏起来，只允许在插件内部进行访问。在下面示例中定义了一个验证函数，这个函数作为 color() 插件的私有函数，是不允许外界访问和修改的，用它来验证用户传递的参数是否合法。对于下面的非法参数设置，则忽略该方法调用，但是不会抛出异常。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
(function($){
    $.extend($.fn,{
        color : function(options){
            if(!filter(options))                //调用隐私方法验证参数，不合法则返回
                return this;
            var options = $.extend({}, $.fn.color.defaults, options);
            return this.each(function(){
                $(this).css("color", options.fcolor);
                $(this).css("backgroundColor", options.bcolor);
                var _html = $(this).html();
                _html = $.fn.color.format(_html);
                $(this).html(_html);
            })
        }
    });
});
```

```

$.fn.color.defaults = {                                //独立设置$.fn.color 对象的默认参数值
    bcolor : "white",
    fcolor : "black"
}
$.fn.color.format = function(str){                      //开放的功能函数
    return str;
};
function filter(options){                               //定义隐私函数，外界无法访问
    //如果参数不存在，或者存在且为对象，则返回 true，否则返回 false
    return !options || (options && typeof options === "object")?true : false;
}
})(jQuery);
$(function(){
    $("p").color("#fff");
})
</script>
<title>上机练习</title>
</head>
<body>
<p>段落文本 1</p>
</body>
</html>

```

4. 让插件具有非破坏性

在特定情况下，jQuery 对象方法可能会修改 jQuery 对象匹配的 DOM 元素，这时就有可能破坏方法的返回值的一致性。为了遵循 jQuery 框架的核心设计理念，应该避免修改 jQuery 对象。

【示例 15】 定义一个 jQuery 对象方法 parent()，获取 jQuery 匹配的所有 DOM 元素的父元素。该方法通过遍历所有匹配元素，获取每个 DOM 元素的父元素，并把这些父元素存储到一个临时数组中，通过过滤、打包，最后返回。

最后，利用自定义的插件 parent() 为所有 p 元素的父元素添加一个边框，示例代码如下所示。演示效果如图 9.12 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
//自定义插件，扩展 jQuery 对象方法，获取所有匹配元素的父元素
(function($){
    $.extend($.fn,{
        parent : function(options){
            var arr = [];
            $.each(this, function(index, value){           //遍历匹配的 DOM 元素
                arr.push(value.parentNode);                 //把匹配元素的父元素推入临时数组
            });
            arr = $.unique(arr);                             //过滤临时数组中重复的元素
            this.length = 0;
            Array.prototype.push.apply( this, arr );       //把变量 arr 打包为伪数组类型返回
            return this;
        }
    });

```



```

    }
  })
})(jQuery);
$(function(){
  var $p = $("p");
  $p.parent().css("border","solid 1px red");
})
</script>
<style type="text/css">
div.big { width:400px; height:400px; }
div.small { width:200px; height:200px; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="big">
  <p> </p>
  <div class="small">
    <p> </p>
  </div>
</div>
</body>
</html>

```

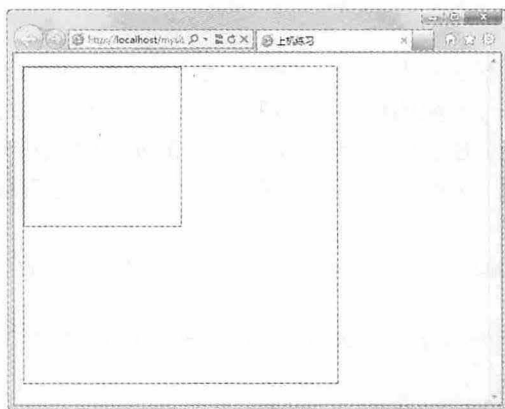


图 9.12 应用自定义插件 parent()

如果在设置了父元素的边框后，希望把与 jQuery 对象匹配的所有元素隐藏起来，可以添加下面代码，则在浏览器中预览就会发现 div 元素也被隐藏起来。

```

$(function(){
  var $p = $("p");
  $p.parent().css("border","solid 1px red");
  $p.hide();
})

```

也就是说，在上面代码中，\$p 变量已经被修改，它不再指向当前 jQuery 对象，而是 jQuery 对象匹配元素的父元素，因此为 \$p 调用 hide() 方法，就会隐藏 div 元素，而不是 p 元素。

这是破坏性操作的一种表现，如果要避免此类行为，建议采用非破坏性操作。例如，在本例中可以使用 pushStack() 方法创建一个新的 jQuery 对象，而不是修改 this 所引用的 jQuery 对象，避免了这种破坏性操作行为，同时 pushStack() 方法还允许调用 end() 方法操作新创建的 jQuery 对象方法，把上面示例的 jQuery 对象方法进行优化。代码如下：

```

(function($){
    $.extend($.fn,{
        parent : function(options){
            var arr = [];
            $.each(this, function(index, value){
                arr.push(value.parentNode);
            });
            arr = $.unique(arr);
            return this.pushStack(arr); //返回新创建的 jQuery 对象，而不是修改后的当前 jQuery 对象
        }
    })
})(jQuery);

```

这时，如果继续执行上面的演示示例操作，则可以看到 div 元素边框样式被定义为红色，同时也隐藏了其包含的 p 元素。也可以采用连续行为进行编写，其中 end() 方法能够恢复被破坏的 jQuery 对象，也就是说，parent() 方法返回的是当前元素的父元素的集合，在调用 end() 方法之后，又恢复到最初的当前元素集合，此时可以继续调用方法作用于原来的 jQuery 对象。代码如下：

```

$(function(){
    var $p = $("p");
    $p.parent().css("border","solid 1px red").end().hide();
})

```

9.3 案例实战：制作 jQuery 文字提示插件

下面通过一个小型实用的 jQuery 插件，帮助读者快速掌握扩展 jQuery 功能的方法和操作步骤。这个插件实现当鼠标滑过链接出现文字提示，提示框的背景颜色可以自由控制。

9.3.1 功能讲解

- ☒ 鼠标经过目标对象时，会显示带有 title 属性的容器，呈现 title 属性值包含的文本内容，目标对象一般是链接、图片等。
- ☒ 可以控制提示框的背景颜色。

9.3.2 构建结构

新建 HTML 文档，构建一个简单的列表导航结构。代码如下：

```

<div>
    <h2>推荐 9 个 jQuery 手风琴菜单插件</h2>
    <ol>
        <li><a href="http://jqueryui.com/demos/accordion/" title="jQuery UI accordion" target="_blank">jQuery UI accordion</a></li>
        <li><a href="http://www.i-marco.nl/weblog/jquery-accordion-3/" title="initMen3.1" target="_blank"> initMen3.1</a></li>
        <li><a href="http://roshanbh.com.np/2008/06/accordion-menu-using-jquery.html" title="Making Accordion Menu Using jQuery">Making Accordion Menu Using jQuery</a></li>
        <li><a href="http://www.lateralcode.com/jquery-accordion-menu/" title="jQuery Accordion Menu">jQuery Accordion Menu</a></li>
        <li><a href="http://www.portalzine.de/index?/Horizontal_Accordion--print" title=" 横 向 手 风 琴 菜 单">Horizontal Accordion Menu</a></li>
    </ol>
</div>

```

```

">jQuery - Horizontal Accordion</a></li>
  <li><a href="http://designreviver.com/tutorials/jquery-examples-horizontal-accordion/" title="简单的横向手风琴效果" class="white">jQuery Examples - Horizontal Accordion</a></li>
  <li><a href="http://berndmatzner.de/jquery/hoveraccordion/" title="Regular and Hover Accordion Menus" class="blue">Regular and Hover Accordion Menus</a></li>
  <li><a href="http://jqueryfordesigners.com/slide-out-and-drawer-effect/" title="Slide Out and Drawer" class="green">Slide Out and Drawer Effect</a></li>
  <li><a href="http://www.scriptocean.com/accordion.html" title="类似 RIA 之家的导航区域的效果" class="red">Javascript Accordion Menu Wizard</a></li>
</ol>
</div>

```

在上面结构中最重要的是 title 属性，该属性用于控制提示框出现的内容。class="blue" 类样式用于控制提示框的背景颜色。当鼠标第 1 次滑过列表项中某个链接对象时，会创建类似下面的提示框容器：

```

<a href="http://berndmatzner.de/jquery/hoveraccordion/" title="Regular and Hover Accordion Menus" class="blue">Regular and Hover Accordion Menus
  <span class="colorTip" style="margin-left: -60px;"> Regular and Hover Accordion Menus
    <span class="pointyTipShadow"></span>
    <span class="pointyTip"></span>
  </span>
</a>

```

9.3.3 设计思路

jQuery 插件设计一般都遵循固定的模板，模板如下：

```

(function($){
  $.fn.插件名 = function(settings){
    //默认参数
    var defaultSettings = {
    }
    /* 合并默认参数和用户自定义参数 */
    settings = $.extend(defaultSettings,settings);
    return this.each(function(){
    });
  }
})(jQuery);

```

这是最基础的 jQuery 插件结构。先来看模板中的第 1 行代码：

```

(function($){
  })(jQuery);

```

这行代码其实用于创建一个匿名函数。读者应该对 JavaScript 闭包有所了解，如果对匿名函数和闭包不了解，会对这种代码非常疑惑，建议阅读 JavaScript 匿名函数及函数的闭包相关知识。

模板中匿名函数的作用是用来保护 \$ 这个变量，避免 \$ 变量与页面中的全局变量冲突。这点非常重要，\$ 变量名在网页脚本中使用率非常高，用户一般无法保证所引入的其他 JavaScript 插件或者代码都使用 \$ 来代表 jQuery。

jQuery 是 jQuery 库定义的一个全局变量，而 \$ 变量名相当于 jQuery 的简写，\$ 冲突率是非常高的，不同的 JavaScript 框架中 \$ 有不同的含义，但如果都使用 jQuery，那又会非常繁琐。上面匿名函数创建了闭包，意味着在这个闭包内，可以任意使用 \$ 变量，不用担心冲突的问题。同时，将 jQuery 这个全局变量传入匿名函数，并执行匿名函数。

在第 2 行代码中，\$.fn 与 jQuery.fn 本质上可以等于 jQuery.prototype。prototype 表示原型继承，prototype

在 JavaScript 中极其重要，是 JavaScript 实现面向对象编程的关键。以 colorTip 为例，代码如下：

```
(function($){
    $.fn.colorTip= function(settings){
        alert(1);
    }
});
```

上面代码实际上给 jQuery 扩展了一个名为 colorTip 的方法，接下来就可以按如下方法调用执行该方法：

```
(function($){
    $('a').colorTip();
});
```

在 \$.fn.colorTip 中，this 上下文就会指向 \$('a') 这个对象。

在接下来的代码中，\$.extend 在 jQuery 插件开发中有很重要的作用，就是合并参数。以 colorTip 为例，先定义默认参数对象 defaultSettings，然后用调用插件时传递的参数覆盖默认参数，实现参数合并。代码如下：

```
(function($){
    var defaultSettings = {
        //颜色
        color      : 'yellow',
        //延迟
        timeout     : 500
    }
    //提示框的颜色
    var supportedColors = ['red','green','blue','white','yellow','black'];
    /* 合并默认参数和用户自定义参数 */
    settings = $.extend(defaultSettings,settings);
});
```

插件调用的代码如下：

```
$('a').colorTip({color:'blue'});
```

如果运行以上代码，就会发现弹出的值为 blue，而不再是默认的 yellow。\$.extend(defaultSettings,settings) 的含义是使用 settings 来覆盖 defaultSettings（同名键值）。实际上，extend() 不止接收两个参数，相对于模板上的写法，还可以按如下方法编写：

```
settings = $.extend({},defaultSettings,settings);
```

即不去覆盖 defaultSettings（默认参数），而是合并到一个空对象上。

接下来增加一个 animate 参数，这个参数也是个对象类型：

```
var defaultSettings = {
    //颜色
    color      : 'yellow',
    //延迟
    timeout     : 500,
    animate     : {type:"fade",speed:"fast"}
}
```

调用如下：

```
$('a').colorTip({color:'yellow',animate:{type:"slide"}});
```

在 settings = \$.extend({},defaultSettings,settings) 下加上：

```
alert(settings.animate.speed);
```

按理说，应该得到的是 fast，实际上却是 undefined。原因是 animate 是对象，不开启深度拷贝，extend() 方法就会直接覆盖。此时可以使用下面代码进行合并：

```
settings = $.extend(true,defaultSettings,settings);
```

```
alert(settings.animate.speed);
```

这样就会得到 fast 返回值。

在最后几行代码中，应该明白下面几个问题：

```
return this.each(function(){
    //代码
});
```

☑ this 指代什么？

调用插件如下：

```
$('[title]').colorTip({color:'yellow'});
```

那么这里 this 实际上是指向 \$('[title]')。

☑ 为什么使用 return？

this.each() 执行完后返回的是 this，这时候再 return this.each()，返回的依旧是 this。而这个 this 上下文又是指代 \$('[title]')，意味着用户可以在 colorTip() 后继续加其他方法。例如：

```
$('[title]').colorTip({color:'yellow'}).size();
```

☑ 为什么要使用 each？

先看下面代码：

```
$('[title]').colorTip({color:'yellow'});
```

\$('[title]') 很明显是一个对象集合，即所有带有 title 属性的容器都能出现提示框，所以需要遍历 \$('[title]') 对象。

9.3.4 难点突破

该插件脚本实现比较简单，难点是如何实现可以自由控制颜色的提示框。要控制提示框主体的边框颜色和背景颜色自然不难，难在如何自由控制三角部分的颜色。通过分析会发现，只有使用纯 CSS 设计的三角，才可以自由控制其颜色。这里就需要用到一个 CSS 技巧：使用 CSS 中的 border 属性设计三角形状。

关于 CSS 设计三角形的技巧细节，这里就不再展开。当然，读者还要思考：如何给三角加上 1 像素的边框。实际上提示框的三角边框不是边框，而是另外一个三角容器。

```
<span class="colorTip" style="margin-left: -60px;"> Regular and Hover Accordion Menus
  <span class="pointyTipShadow"></span>
  <span class="pointyTip"></span>
</span>
```

其中， 和 是两个空的标签，它们的作用正是用来设计三角形的。 三角宽度为 12px，颜色与提示框主体背景颜色一样，而 三角宽度为 14px，放在 的下面，颜色与提示框主体边框颜色一样，就产生了边框错觉。

然后通过 CSS 设计三角样式，代码如下：

```
.colorTip { display:none; position:absolute; left:50%; top:-30px; padding:6px 12px; background-color:white;
font-size:12px; font-style:normal; line-height:1; text-decoration:none; text-align:center; text-shadow:0 0 1px
white; white-space:nowrap; -moz-border-radius:4px; -webkit-border-radius:4px; border-radius:4px; }
.pointyTip, .pointyTipShadow { border:6px solid transparent; bottom:-12px; height:0; left:50%; margin-left:-6px;
position:absolute; width:0; }
.pointyTipShadow { border-width:7px; bottom:-14px; margin-left:-7px; }
.colorTipContainer { position:relative; text-decoration:none !important; }
```

代码看上去不少，其实只要理解 border 生成三角就可以了，剩下的代码主要还是用于调整位置。整个 CSS 样式代码如下：

```
<style type="text/css">
/* 插件样式 */
.colorTip { display:none; position:absolute; left:50%; top:-30px; padding:6px 12px; background-color:white;
font-size:12px; font-style:normal; line-height:1; text-decoration:none; text-align:center; text-shadow:0 0 1px
```

```

white; white-space:nowrap; -moz-border-radius:4px; -webkit-border-radius:4px; border-radius:4px; }
.pointyTip, .pointyTipShadow { border:6px solid transparent; bottom:-12px; height:0; left:50%; margin-left:-6px;
position:absolute; width:0; }
.pointyTipShadow { border-width:7px; bottom:-14px; margin-left:-7px; }
.colorTipContainer { position:relative; text-decoration:none !important; }
/* 6 个不同颜色的模板 */
.white .pointyTip { border-top-color:white; }
.white .pointyTipShadow { border-top-color:#ddd; }
.white .colorTip { background-color:white; border:1px solid #DDDDDD; color:#555555; }
.yellow .pointyTip { border-top-color:#f9f2ba; }
.yellow .pointyTipShadow { border-top-color:#e9d315; }
.yellow .colorTip { background-color:#f9f2ba; border:1px solid #e9d315; color:#5b5316; }
.blue .pointyTip { border-top-color:#d9f1fb; }
.blue .pointyTipShadow { border-top-color:#7fcdee; }
.blue .colorTip { background-color:#d9f1fb; border:1px solid #7fcdee; color:#1b475a; }
.green .pointyTip { border-top-color:#f2fdf1; }
.green .pointyTipShadow { border-top-color:#b6e184; }
.green .colorTip { background-color:#f2fdf1; border:1px solid #b6e184; color:#558221; }
.red .pointyTip { border-top-color:#bb3b1d; }
.red .pointyTipShadow { border-top-color:#8f2a0f; }
.red .colorTip { background-color:#bb3b1d; border:1px solid #8f2a0f; color:#f9f2ba; text-shadow:none; }
.black .pointyTip { border-top-color:#333; }
.black .pointyTipShadow { border-top-color:#111; }
.black .colorTip { background-color:#333; border:1px solid #111; color:#f9f2ba; text-shadow:none; }
</style>

```

9.3.5 代码实现

解决了颜色控制这个难点，接下来代码实现难度就不大了，详细说明如下。

1. 设置参数

本插件需要提供两个参数：颜色和隐藏提示框的延迟时间。当然可以根据需要提供更多的配置参数。

```

var defaultSettings = {
    color      : 'yellow',    //颜色
    timeout    : 500         //延迟
}
/* 合并默认参数和用户自定义参数 */
settings = $.extend(defaultSettings,settings);

```

2. 创建数组

创建一个包含所有颜色信息的数组。

```

//提示框的颜色
var supportedColors = ['red','green','blue','white','yellow','black'];

```

3. 设计定时器

本插件需要定义一个定时器，这个定时器用于定义鼠标移开目标容器多长时间隐藏提示框。

```

/*定时器类*/
function eventScheduler(){
    eventScheduler.prototype = {
        set : function (func,timeout){    //添加定时器
            this.timer = setTimeout(func,timeout);
        }
    }
}

```

```

    },
    clear: function(){           //清理定时器
        clearTimeout(this.timer);
    }
}

```

eventScheduler 类结构简单，该类型包含两个原型方法：set()用来添加定时器，clear()用来清理定时器。

4. 设计提示框

创建提示框类 Tip，该类型包含两个本地属性：content 和 shown。其中 content 定义提示框包含的文本内容，shown 表示一个开关按钮，定义是否显示提示框。同时，该类型还包含 3 个原型方法：generate()、show() 和 hide()。

```

/*提示类*/
function Tip(txt){
    this.content = txt;
    this.shown = false;
}
Tip.prototype = {
    generate: function(){           //产生提示框
        return this.tip || (this.tip = $('<span class="colorTip">'+this.content+'<span class="pointyTipShadow">
</span><span class="pointyTip"></span></span>'));
    },
    show: function(){              //显示提示框
        if(this.shown) return;
        this.tip.css('margin-left',-this.tip.outerWidth()/2).fadeIn('fast');
        this.shown = true;
    },
    hide: function(){              //隐藏提示框
        this.tip.fadeOut();
        this.shown = false;
    }
}

```

注意，Tip 和 eventScheduler 类都是在 \$.fn.colorTip 函数体外定义的，可以自由调用，但是它们都是私有类型，外界是无法访问的。

5. 设计 Colortip 代码

首先，需要实例化 Tip 和 eventScheduler 类，代码如下：

```

//实例化 eventScheduler（定时器）
var scheduleEvent = new eventScheduler();
//实例化 Tip（提示类，产生、显示、隐藏）
var tip = new Tip(elem.attr('title'));

```

然后，产生提示框，将提示框加入到目标容器，并给提示框父容器添加样式，代码如下：

```
elem.append(tip.generate()).addClass('colorTipContainer');
```

再检查提示框父容器是否有颜色样式，这里只需给没有颜色样式的目标容器加入默认的颜色样式（黄色），代码如下：

```

var hasClass = false;
for(var i=0;i<supportedColors.length;i++){
    if(elem.hasClass(supportedColors[i])){
        hasClass = true;
        break;
    }
}

```

```
// 如果没有，使用默认的颜色
if(!hasClass){
    elem.addClass(settings.color);
}
```

最后，给目标容器添加鼠标滑过事件。设计当鼠标滑过提示框父容器时，显示提示框，鼠标移出，则隐藏。代码如下：

```
elem.hover(function(){
    tip.show();
    //清理定时器
    scheduleEvent.clear();
},function(){
    //启动定时器
    scheduleEvent.set(function(){
        tip.hide();
    },settings.timeout);
});
//删除 title 属性
elem.removeAttr('title');
```

至此，Colortip 插件代码设计全部结束，该插件能够自由控制提示框颜色。但 Colortip 还存在不少的局限性，只能满足基本的应用，这里仅作为学习案例帮助读者快速掌握 jQuery 插件的一般开发过程。整个插件的完整代码如下：

```
(function($){
    $.fn.colorTip = function(settings){
        var defaultSettings = {
            color      : 'yellow',
            timeout     : 500
        }
        var supportedColors = ['red','green','blue','white','yellow','black'];
        settings = $.extend(defaultSettings,settings);
        return this.each(function(){
            var elem = $(this);
            if(!elem.attr('title')) return true;
            var scheduleEvent = new eventScheduler();
            var tip = new Tip(elem.attr('title'));
            elem.append(tip.generate()).addClass('colorTipContainer');
            var hasClass = false;
            for(var i=0;i<supportedColors.length;i++){
                if(elem.hasClass(supportedColors[i])){
                    hasClass = true;
                    break;
                }
            }
            if(!hasClass){
                elem.addClass(settings.color);
            }
            elem.hover(function(){
                tip.show();
                scheduleEvent.clear();
            },function(){
                scheduleEvent.set(function(){
                    tip.hide();
                },settings.timeout);
            });
        });
    };
});
```



```

        elem.removeAttr('title');
    });
}
function eventScheduler(){}
eventScheduler.prototype = {
    set      : function (func,timeout){
        this.timer = setTimeout(func,timeout);
    },
    clear: function(){
        clearTimeout(this.timer);
    }
}
function Tip(txt){
    this.content = txt;
    this.shown = false;
}
Tip.prototype = {
    generate: function(){
        return this.tip || (this.tip = $('<span class="colorTip">'+this.content+'<span class="pointyTipShadow"></span> <span class="pointyTip"></span></span>'));
    },
    show: function(){
        if(this.shown) return;
        this.tip.css('margin-left',-this.tip.outerWidth()/2).fadeIn('fast');
        this.shown = true;
    },
    hide: function(){
        this.tip.fadeOut();
        this.shown = false;
    }
}
})(jQuery);

```

9.3.6 应用插件

完成插件的设计，就可以在页面中应用该插件了，使用 jQuery 匹配文档中包含 title 属性的元素，然后调用 colorTip() 方法即可。在调用时可以传递参数对象，设置个人的提示框字体颜色和背景颜色等样式。案例演示效果如图 9.13 所示。

```

$(function(){
    $('[title]').colorTip({color:'yellow'});
})

```



图 9.13 应用自定义插件 colorTip()

第10章

表格开发

( 视频讲解：1 小时 10 分钟)

随着 Web 标准普及，基于表格的页面布局已经逐步被基于 CSS 的设计所取代，虽然表格在 90 年代是用来创建多栏或者其他复杂布局的常用工具而且是必要的权宜之计，但是设计表格的目的却完全不是为了进行页面布局。

本章不讨论表格的角色问题，主要讲解如何使用 jQuery 来增强表格的可读性、可用性以及视觉冲击力，如何显示并与作为表格式数据语义容器的表格进行交互，探讨如何为表格应用具有语义的、可访问性的 HTML 标记。

10.1 数据排序

数据排序是表格操作的常见任务，在大型数据表格中，能够对搜索的信息进行排序是非常重要的，表格结构具有天然的序列化规律，这个操作实现的方法和技巧也是多样的。实际开发中，表格排序的实现途径主要有两种：一种是在数据生成时由服务器负责进行排序，另一种是在数据显示时通过 JavaScript 脚本在客户端进行动态排序。更多时候，用户还是希望直接使用 JavaScript 进行排序，这样既不需要服务端的响应等待，也不需要考虑服务器端的操作权限。

10.1.1 构建符合数据排序的表格结构

根据常规设计习惯，当用户单击数据表格的列标题时，表格能够自动根据该列的数据进行排序，因此在开发之前，用户需要考虑以下 3 个问题：

- ☑ 把表格的列标题作为按钮，通过为该按钮绑定 click 事件，以便触发排序函数，实现按照相应的列进行排序。
- ☑ 使用表格的<thead>和<tbody>标签把数据分割为行组，以方便 JavaScript 脚本进行有针对性的控制，避免把列标题或者其他辅助行信息混淆在排序序列中。很多读者可能忽略了这两个标签的作用，其实添加这两个标签有助于我们更方便地使用 CSS 选择器。
- ☑ 构建符合数据排序的表格结构既要考虑表格的扩展性，还要考虑方法的通用性。在动态数据表格中，无法预知表格数据的长度和宽度，也无法预知用户对表格的额外要求，如添加表格页脚，对数据进行分组。另外，还要确保表格在不同的网页环境中都能够正确显示和有效交互。

为此，在这里构建了一个简单的表格结构，通过<thead>和<tbody>标签对数据行进行分组，避免数据行

和标题行的混淆。同时通过<th>和<td>标签有效减少单元格互用。代码如下：

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>产品名称</th>
      <th>标准成本</th>
      <th>列出价格</th>
      <th>单位数量</th>
      <th>最小再订购数量</th>
      <th>类别</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td>苹果汁</td>
      <td>5.00</td>
      <td>30.00</td>
      <td>10 箱 x 20 包</td>
      <td>10</td>
      <td>饮料</td>
    </tr>
    <tr>
      <td>3</td>
      <td>蕃茄酱</td>
      <td>4.00</td>
      <td>20.00</td>
      <td>每箱 12 瓶</td>
      <td>25</td>
      <td>调味品</td>
    </tr>
    ...
    <tr>
      <td>81</td>
      <td>绿茶</td>
      <td>4.00</td>
      <td>20.00</td>
      <td>每箱 20 包</td>
      <td>25</td>
      <td>饮料</td>
    </tr>
  </tbody>
</table>
```

其中，正文数据中雷同的结构和数据，考虑到篇幅的问题，上面代码就没有全部显示。

在网页头部区域添加<style>标签，定义内部样式表，对数据表格适当进行美化。其中要考虑几个常用类样式的设计工作。

- ☑ td.sorted: 设计排序列单元格的背景色，以便高亮显示排序列。
- ☑ th.sorted-asc: 设计排序列标题单元格箭头提示的背景图像标识，提示升序排序。
- ☑ th.sorted-desc: 设计排序列标题单元格箭头提示的背景图像标识，提示降序排序。
- ☑ tr.even, tr.first: 设计隔行换色的显示样式，即单行背景样式。

☑ tr.odd, tr.second: 设计隔行换色的显示样式, 即双行背景样式。

☑ tr.third: 设计特殊行背景样式。

详细代码如下:

```
<style type="text/css">
table { font-size:12px; width:100%; table-layout:fixed; empty-cells:show; border-collapse: collapse; margin:0
auto; border:1px solid #cad9ea; color:#666; }
th { background-image: url(images/th_bg1.gif); background-repeat:repeat-x; height:30px; cursor:pointer; }
td { height:20px; }
td, th { border:1px solid #cad9ea; padding:0 1em 0; }
td.sorted { background: #ffd; }
th.sorted-asc { background: url('images/icons/arrow_up.png') no-repeat 0 50%; }
th.sorted-desc { background: url('images/icons/arrow_down.png') no-repeat 0 50%; }
tr.even, tr.first { background-color: #fff; }
tr.odd, tr.second { background-color: #f5f5f5; }
tr.third { background-color: #ccc; }
</style>
```

初步设计后的表格样式效果如图 10.1 所示。

ID	产品名称	标准成本	市场价值	单位数量	最小订购数量	类别
1	菜肉干	5.00	30.00	10箱 * 20包	10	饮料
3	葡萄干	4.00	10.00	每箱10包	20	调味品
4	盐	3.00	25.00	每箱10包	10	调味品
5	洋葱	12.00	50.00	每箱10包	10	调味品
6	葡萄干	5.00	10.00	每箱10包	20	调味品
7	菜肉干	5.00	10.00	每箱10包	10	调味品
8	葡萄干	15.00	30.00	每箱10包	10	调味品
14	沙茶	12.00	20.00	每箱10包	10	调味品
17	猪肉	3.00	9.00	每箱10包	10	调味品
19	香酥	10.00	25.00	每箱10包	5	调味品
24	绿豆芽	25.00	60.00	每箱10包	10	调味品
27	花生	15.00	30.00	每箱10包	5	调味品
28	鸡蛋	10.00	10.00	每箱10包	15	调味品
40	鸡蛋	5.00	25.00	每箱10包	30	调味品
41	鸡蛋	8.00	20.00	每箱10包	10	调味品
43	鸡蛋	10.00	20.00	每箱10包	25	调味品
46	玉米片	5.00	15.00	每箱10包	25	调味品
51	猪肉干	15.00	40.00	每箱10包	10	调味品
52	芝麻一透片	12.00	30.00	每箱10包	25	调味品
58	猪肉	3.00	10.00	每箱10包	30	调味品
67	小羊	4.00	12.00	每箱10包	10	调味品
68	鸡蛋	8.00	20.00	每箱10包	10	调味品
68	鸡蛋	10.00	25.00	每箱10包	10	调味品
72	猪肉干	5.00	8.00	每箱10包	10	调味品
74	鸡蛋	8.00	15.00	每箱10包	5	调味品
77	猪肉干	2.00	10.00	每箱10包	15	调味品
78	猪肉干	5.00	10.00	每箱10包	25	调味品

图 10.1 设计的表格样式

10.1.2 JavaScript 的基本排序方法

在实际进行排序时, 可以使用 JavaScript 内置方法。Array 对象定义了两个方法来调整数组顺序。

☑ reverse(): 颠倒数组中元素的顺序。

☑ sort(): 对数组元素进行排序。

1. reverse()方法

reverse()是一个简单的数组元素调整方法, 它能够颠倒数组元素的排列顺序, 就是倒序处理数组, 该方法不需要参数。例如:

```
var a = [1,2,3,4,5];           //定义数组
a.reverse();                   //颠倒数组顺序
alert(a);                      //返回数组[5,4,3,2,1]
```


注意，该方法是在原数组基础上进行操作，而不是创建新的数组。

2. sort()方法

sort()方法与 reverse()方法不同，它不是简单的颠倒顺序，而是根据一定的条件对数组元素进行排序。如果调用 sort()方法时没有传递参数，则按字母顺序对数组中的元素进行排序。例如：

```
var a = ["a","e","d","b","c"];           //定义数组
a.sort();                                 //按字母顺序对元素进行排序
alert(a);                                 //返回数组[a,b,c,d,e]
```

使用该方法时，应该注意以下几个问题：

- ☑ 字母顺序就是字母在字符编码表中的顺序，每个字符在字符表中都有一个唯一的编号。
- ☑ 如果元素不是字符串，则 sort()方法会试图把数组的元素都转换成字符串，以便进行比较。
- ☑ 在排序时，sort()方法将根据元素值进行逐位比较，而不是根据字符串的个数或整体状况进行排序。

例如：

```
var a = ["aba","baa","aab"];           //定义数组
a.sort();                                 //按字母顺序对元素进行排序
alert(a);                                 //返回数组[aab,aba,baa]
```

在排序时，首先比较每个元素的第 1 个字符，在第 1 个字符相同的情况下，再比较第 2 个字符，依此类推。

- ☑ 在任何情况下，数组中 undefined 的元素都排列在数组末尾。
- ☑ sort()方法是在原数组基础上进行排序操作的，不会创建新的数组。

当然，sort()方法不仅仅按字母顺序进行排序，还可以根据其他顺序执行操作。这时就必须为方法提供一个比较函数的参数，该函数要比较两个值，然后返回一个用于说明这两个值的相对顺序的数字。比较函数应该具有两个参数 a 和 b，其返回值如下：

- ☑ 如果根据自定义评判标准，a 小于 b，在排序后的数组中，a 应该出现在 b 之前，就返回一个小于 0 的值。
- ☑ 如果 a 等于 b，就返回 0。
- ☑ 如果 a 大于 b，就返回一个大于 0 的值。

例如，在下面的示例中，将根据比较函数比较数组中每个元素的大小，并按从小到大的顺序执行排序：

```
function f( a, b ){                       //比较函数
    return ( a - b )                      //返回比较参数
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9];   //定义数组
a.sort(f);                               //根据数字大小由小到大进行排序
alert( a );                              //返回数组[0,1,2,3,4,5,6,7,8,9]
```

如果按从大到小的顺序执行排序，则可以让返回值取反即可。代码如下：

```
function f( a, b ){                       //比较函数
    return -( a - b )                    //取反并返回比较参数
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9];   //定义数组
a.sort(f);                               //根据数字大小由大到小进行排序
alert( a );                              //返回数组[9,8,7,6,5,4,3,2,1,0]
```

sort()方法用法比较灵活，但更灵活的是对比较函数的设计。例如，如果根据奇偶数顺序排列数组，用户只需要判断比较函数中两个参数是否为奇偶数，并决定排列顺序。代码如下：

```
function f( a, b ){                       //比较函数
    var a = a % 2;                        //获取参数 a 的奇偶性
    var b = b % 2;                        //获取参数 b 的奇偶性
    if( a == 0 ) return 1;                //如果参数 a 为偶数，则排在左边
```

```

    if( b == 0 ) return -1;           //如果参数 b 为偶数，则排在右边
}
var a = [3, 1, 2, 4, 5, 7, 6, 8, 0, 9]; //定义数组
a.sort( f );                         //根据数字大小由大到小进行排序
alert( a );                          //返回数组[3,1,5,7,9,0,8,6,4,2]

```

sort()方法在调用比较函数时，把每个元素值传递给比较函数。如果元素值为偶数，则保留其位置不动；如果元素值为奇数，则调换参数 a 和 b 的显示顺序，从而实现对数组中所有元素执行奇偶排序。如果希望偶数排在前面，奇数排在后面，则只需要返回值取反。比较函数如下：

```

function f( a, b ){
    var a = a % 2;
    var b = b % 2;
    if( a == 0 ) return -1;
    if( b == 0 ) return 1;
}

```

在正常情况下，对字符串进行排序是区分大小写的，这是因为每个大写和小写字母在字符编码表中的顺序是不同的，大写字母大于小写字母。例如：

```

var a = ["aB", "Ab", "Ba", "bA"]; //定义数组
a.sort();                          //默认方法排序
alert( a );                        //返回数组["Ab","Ba","aB","bA"]

```

也就是说，大写字母总是排在左侧，而小写字母总是排在右侧。如果让小写字母总是排在前面，则可以这样设计：

```

function f( a, b ){                //比较函数，如果 a 小于 b，则 a、b 位置不动，反之换位
    return ( a < b );
}
var a = ["aB", "Ab", "Ba", "bA"];
a.sort( f );                      //根据比较函数进行排序
alert( a );                      //返回数组["Ab","Ba","aB","bA"]

```

对于字母比较大小时，JavaScript 是根据字符编码大小来决定的。当为 true 时，则返回 1；为 false 时，则返回-1。如果不希望区分字母大小，也就是说，大写字母和小写字母按相同顺序排列，则可以这样设计：

```

function f( a, b ){                //比较函数
    var a = a.toLowerCase();       //转换为小写形式
    var b = b.toLowerCase();       //转换为小写形式
    if( a < b ){                   //如果 a 的编码小于 b，则换位操作
        return 1;
    }
    else{                          //否则，保持原位不动
        return -1;
    }
}
var a = ["aB", "Ab", "Ba", "bA"]; //定义数组
a.sort( f );                      //执行排序
alert( a );                      //返回数组["aB", "Ab", "Ba", "bA"]

```

如果调整排序顺序，则为返回值取反即可。

经常会遇到把浮点数和整数分开排列的情况。当然借助 sort()方法，实现起来并不是很难，可以这样设计：

```

function f( a, b ){                //比较函数
    if( a > Math.floor( a ) ) return 1; //如果 a 是浮点数，则调换位置
    if( b > Math.floor( b ) ) return -1; //如果 b 是浮点数，则调换位置
}
var a = [3.55555, 1.23456, 3, 2.11111, 5, 7, 3]; //定义数组
a.sort( f );                          //进行筛选

```

```
alert( a ); //返回数组[3,5,7,3,2.11111,1.23456,3.55555]
```

如果调整排序顺序,则为返回值取反即可。

sort()方法的功能是非常强大的,如果比较的元素是对象而不是值类型(如数字和字符串等)这样简单的数据时,排序就变得更加有趣了。

10.1.3 实现表格基本排序

在对表格进行排序之前,应该注意以下两个问题:

- ☑ 并不是页面中所有表格都需要排序。
- ☑ 并不是表格中每列都需要排序。

因此,在设计之初我们应该为排序表格做一个标记,以便 JavaScript 捕获。同样也应为列进行标记,以便 JavaScript 进行处理。一般可以为表格设计一个排序类进行标识,对于需要排序的列添加一个类标记。代码如下:

```
<table class="sortable">
  <thead>
    <tr>
      <th class="sort-alpha">ID</th>
      <th class="sort-alpha">产品名称</th>
      <th class="sort-alpha">标准成本</th>
      <th class="sort-alpha">列出价格</th>
      <th class="sort-alpha">单位数量</th>
      <th class="sort-alpha">最小再订购数量</th>
      <th class="sort-alpha">类别</th>
    </tr>
  </thead>
  ...
</table>
```

以<table class="sortable">和<th class="sort-alpha">标签为标记,就可以添加脚本实现基本的排序功能了,代码如下:

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$.fn.alternateRowColors = function() {
  $('tbody tr:odd', this).removeClass('even').addClass('odd');
  $('tbody tr:even', this).removeClass('odd').addClass('even');
  return this;
};
$(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $table.alternateRowColors();
    $table.find('th').each(function(column) {
      if($(this).is('.sort-alpha')) {
        $(this).addClass('clickable').hover(function() {
          $(this).addClass('hover');
        }, function() {
          $(this).removeClass('hover');
        }).click(function() {
          var rows = $table.find('tbody > tr').get();
          rows.sort(function(a, b) {
            var a = $(a).children("td").eq(column).text().toUpperCase();
            var b = $(b).children("td").eq(column).text().toUpperCase();
            return a < b ? -1 : a > b ? 1 : 0;
          });
          $table.find('tbody').empty().append(rows);
        });
      }
    });
  });
});
```

```

        var b = $(b).children("td").eq(column).text().toUpperCase();
        if(a < b)
            return -1;
        if(a > b)
            return 1;
        return 0;
    });
    $.each(rows, function(index, row) {
        $table.children('tbody').append(row);
    });
    }
    });
}
</script>

```

在实现数据排序功能之前,用户不妨先为jQuery扩展一个简单的功能插件,这个插件`alternateRowColors()`作为jQuery命令而存在。事实上,任何应用到一组DOM元素上的操作都可以简单地通过插件的形式来表示。通过插件形式调用有以下3个好处:

- ☑ 此时函数是作为jQuery.fn的一个新属性定义的,而非一个孤立的函数,比较方便调用。
- ☑ 可以灵活使用this关键字,此时this引用的是调用该方法的jQuery对象。
- ☑ 在函数的末尾返回this关键字,返回这个值之后,可以使这个新方法能够再连缀其他方法。

在上面代码中使用`each()`方法进行显式迭代,而不是直接使用`$(table.sortable th.sort-alpha).click()`选择并为每个带有`sort-alpha`类的标题单元绑定单击事件处理程序。由于`each()`方法会向它的回调函数中传递迭代索引,所以能够用它方便地捕获到一个关键信息,即单击标题的列索引,进而能够在后面使用这个列索引来找到每个数据行中相关单元格。

在找到带有`sort-alpha`类的标题单元之后,接下来取得一个包含所有数据行的数组,这是一个通过`get()`方法将jQuery对象转换为一个DOM节点的数组。之所以要进行这样的转换,是因为虽然jQuery对象在多方面与数组类似,但是它不具有任何本地数组的方法,如`sort()`方法。

调用`sort()`方法比较简单,它通过比较相关单元格的文本,实现对表格行进行排序,这里主要根据`each()`方法的回调函数中参数可以传递th在table中的列序号,并通过这个列序号获取该列的tbody包含的该列单元格来实现。考虑到文本大小写问题,在比较时应该区分大小写。最后,通过循环遍历排序后的数组,将表格行重新插入到表格中。注意,因为`append()`方法不会复制节点,而是移动表格行,因此可以看到表格数据行重新排序。

由于在排序过程中表格行顺序被打乱重新显示,最初设计的隔行换色的样式发生了变化,如图10.2所示。因此,当完成表格数据行排序之后,应该重新设置隔行换色的背景样式,重新调用`alternateRowColors()`方法即可。代码如下:

```

$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};
$(function() {
    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors();
        $table.find('th').each(function(column) {
            if($(this).is('.sort-alpha')) {

```



```

$(this).addClass('clickable').hover(function() {
    $(this).addClass('hover');
}, function() {
    $(this).removeClass('hover');
}).click(function() {
    var rows = $table.find('tbody > tr').get();
    rows.sort(function(a, b) {
        var a = $(a).children("td").eq(column).text().toUpperCase();
        var b = $(b).children("td").eq(column).text().toUpperCase();
        if(a < b)
            return -1;
        if(a > b)
            return 1;
        return 0;
    });
    $.each(rows, function(index, row) {
        $table.children('tbody').append(row);
    });
    $table.alternateRowColors();
});
}
});
});

```

ID	产品名称	标准成本	列出价格	单位数量	最小起订数量	类别
52	三合一麦片	12.00	30.00	每箱24包	25	谷类/麦片
34	啤酒	10.00	30.00	每箱24瓶	15	饮料
57	小米	4.00	12.00	每袋5公斤	20	意大利面类
42	柳橙汁	10.00	30.00	每箱24瓶	25	饮料
20	桂花糕	25.00	60.00	每箱30盒	10	糕点
14	沙茶	12.00	30.00	每箱12瓶	10	干果和坚果
65	海苔卷	8.00	30.00	每箱24瓶	10	调味品
7	海鲜粉	20.00	40.00	每箱30盒	10	干果和坚果
17	猪肉	2.00	9.00	每袋500克	10	水果和蔬菜罐头
34	猪肉干	15.00	40.00	每箱24包	10	干果和坚果
49	玉米片	5.00	15.00	每箱24包	25	点心
50	白茶	3.00	10.00	每袋5公斤	30	意大利面类
4	盐	8.00	25.00	每箱12瓶	10	调味品
19	雪梨	10.00	45.00	每箱30盒	5	粉类食品
31	榨菜	4.00	20.00	每箱30包	25	饮料
26	肉松	10.00	35.00	每箱24瓶	20	调味品
9	胡椒粉	15.00	35.00	每箱30盒	10	调味品
21	花生	15.00	35.00	每箱30包	5	粉类食品
1	苹果汁	5.00	30.00	19瓶 = 20瓶	10	饮料
30	葡萄干	2.00	10.00	每包500克	25	干果和坚果
5	紫菜	2.00	20.00	每包100克	20	调味品

图 10.2 初步实现的排序效果

10.1.4 优化排序性能

直接调用 JavaScript 的 `sort()` 方法进行排序，运行速度比较慢，特别是表格数据比较多时，让人难以忍受。因为在执行 `sort()` 方法时，会反复调用参数函数进行比较，这个处理过程工作量非常大，JavaScript 使用的排序算法在标准中没有说明，可能是一种较慢的冒泡排序算法，这种算法速度是非常慢的。

要解决这个性能问题，不妨预先计算用于比较的关键字，可以提取每个排序单元格中的关键字计算，

并将这个过程从迭代回调函数中抽离出来，在一个单独的循环中完成，避免在回调函数中被反复调用。代码如下：

```
var rows = $table.find('tbody > tr').get();
$.each(rows, function(index, row) {
    row.sortKey = $(row).children('td').eq(column).text().toUpperCase();
});
rows.sort(function(a, b) {
    if(a.sortKey < b.sortKey)
        return -1;
    if(a.sortKey > b.sortKey)
        return 1;
    return 0;
});
$.each(rows, function(index, row) {
    $table.children('tbody').append(row);
    row.sortKey = null;
});
```

在一个循环中，把所有占用资源的工作全部完成，并把计算的结果保存到每个单元格的新属性中，这个属性并非是 DOM 预定义属性，但是考虑到每个单元格都需要这样一个关键字，通过属性的方式保存，当调用回调函数进行比较时，可以直接读取每个单元格的这个新属性值，避免重复计算。

当完成排序操作之后，应该删除 sortKey 属性，以便手动释放内存，避免大量的 sortKey 属性值占用系统资源，导致内存泄露。

10.1.5 设计其他类型排序

sort() 默认排序方式是根据字符编码顺序进行计算，当然，对于不同类型的数据可能希望采用其他类型排序方式，如日期、数字、货币等。根据这些数据类型的特点，可以在关键字计算中进行处理。代码如下：

```
$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};
$(function() {
    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors();
        $table.find('th').each(function(column) {
            var findSortKey;
            if($(this).is('.sort-alpha')) {
                findSortKey = function($cell) {
                    return $cell.text().toUpperCase();
                };
            } else if($(this).is('.sort-numeric')) {
                findSortKey = function($cell) {
                    var key = parseFloat($cell.text().replace(/^[^d.]*$/, ""));
                    return isNaN(key) ? 0 : key;
                };
            } else if($(this).is('.sort-date')) {
                findSortKey = function($cell) {

```

```

        return Date.parse('1 ' + $cell.text());
    };
}
if(findSortKey) {
    var rows = $table.find('tbody > tr').get();
    $(this).addClass('clickable').hover(function() {
        $(this).addClass('hover');
    }, function() {
        $(this).removeClass('hover');
    }).click(function() {
        $.each(rows, function(index, row) {
            row.sortKey = findSortKey($(row).children('td').eq(column));
        });
        rows.sort(function(a, b) {
            if(a.sortKey < b.sortKey)
                return -1;
            if(a.sortKey > b.sortKey)
                return 1;
            return 0;
        });
        $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
        });
        $table.alternateRowColors().trigger('repaginate');
    });
}
});
});
}

```

对于货币数据，在比较之前应该去掉货币前缀符号，然后再根据需要进行比较计算。而对于数字类型，需要使用 `parseFloat()` 把值进行类型转换。如果不能够转换，则需要使用 `isNaN()` 方法检测是否为非数字值，然后把它替换为数字 0，避免 NaN 值对 `sort()` 函数造成错误。对于日期类型，由于表格中包含的值不完整，需要把日期格式补充完整。

最后，根据列数据类型在表格的列标题结构中添加排序的类标识。代码如下：

```

<table class="sortable">
  <thead>
    <tr>
      <th class="sort-numeric">ID</th>
      <th class="sort-alpha">产品名称</th>
      <th class="sort-numeric">标准成本</th>
      <th class="sort-numeric">列出价格</th>
      <th class="sort-alpha">单位数量</th>
      <th class="sort-numeric">最小再订购数量</th>
      <th class="sort-alpha">类别</th>
    </tr>
  </thead>
</table>

```

10.1.6 完善排序交互的视觉效果

良好的视觉体验应该对数据表格的动态排序进行提示，只有这样用户才能觉察到数据排序已经发生了变化。这里应该注意以下两个问题：

- ☑ 应该即时标识排序的列和排序的方式。
- ☑ 应该对排序列数据进行高亮显示，以方便用户阅读。

根据上述思考，可以通过突出显示最近用于排序的列，把用户的注意力吸引到很可能包含相关信息的表格部分。既然已经知道了当前列在表格中的位置，因此只需要为当前列单元格添加一个样式类即可。核心代码如下：

```
$table.find('td').removeClass('sorted').filter(':nth-child(' + (column + 1) + ')').addClass('sorted')
```

在上面代码中，首先清除表格中所有单元格中包含的 `sorted` 样式类，然后为当前列单元格添加 `sorted` 样式类，注意列序号的调用。

与排序有关的一个重要视觉设计，就是列数据的升序和降序。降序和升序可以在 `sort()` 方法中的回调函数中进行切换，只需要改变返回值即可。这里通过一个方向变量进行动态控制：

```
rows.sort(function(a, b) {
    if(a.sortKey < b.sortKey)
        return -newDirection;
    if(a.sortKey > b.sortKey)
        return newDirection;
    return 0;
});
```

如果 `newDirection` 等于 1，则按正常的排序方式进行排序；如果等于 -1，则切换排序方式，实现降序排列。然后在代码初始化中对该变量进行初始化声明，并适当与列标题的 `sorted-asc` 样式类进行绑定。代码如下：

```
var newDirection = 1;
if($(this).is('.sorted-asc')) {
    newDirection = -1;
}
```

在排序处理之后，再根据这个临时变量，为列标题添加对应的样式类，同时应该清理掉其他列中绑定的升降样式类。代码如下：

```
$table.find('th').removeClass('sorted-asc').removeClass('sorted-desc');
var $sortHead = $table.find('th').filter(':nth-child(' + (column + 1) + ')');
if(newDirection == 1) {
    $sortHead.addClass('sorted-asc');
} else {
    $sortHead.addClass('sorted-desc');
}
```

最后，整个数据表格排序的完整脚本代码如下：

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript">
$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};
$(function() {
    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors();
```



```

$table.find('th').each(function(column) {
    var findSortKey;
    if($(this).is('.sort-alpha')) {
        findSortKey = function($cell) {
            return $cell.find('.sort-key').text().toUpperCase() + ' ' + $cell.text().toUpperCase();
        };
    } else if($(this).is('.sort-numeric')) {
        findSortKey = function($cell) {
            var key = parseFloat($cell.text().replace(/^[^d.]*/, ""));
            return isNaN(key) ? 0 : key;
        };
    } else if($(this).is('.sort-date')) {
        findSortKey = function($cell) {
            return Date.parse('1 ' + $cell.text());
        };
    }
    if(findSortKey) {
        $(this).addClass('clickable').hover(function() {
            $(this).addClass('hover');
        }, function() {
            $(this).removeClass('hover');
        }).click(function() {
            var newDirection = 1;
            if($(this).is('.sorted-asc')) {
                newDirection = -1;
            }
            var rows = $table.find('tbody > tr').get();
            $.each(rows, function(index, row) {
                row.sortKey = findSortKey($(row).children('td').eq(column));
            });
            rows.sort(function(a, b) {
                if(a.sortKey < b.sortKey)
                    return -newDirection;
                if(a.sortKey > b.sortKey)
                    return newDirection;
                return 0;
            });
            $.each(rows, function(index, row) {
                $table.children('tbody').append(row);
                row.sortKey = null;
            });
            $table.find('th').removeClass('sorted-asc').removeClass('sorted-desc');
            var $sortHead = $table.find('th').filter(':nth-child(' + (column + 1) + ')');
            if(newDirection == 1) {
                $sortHead.addClass('sorted-asc');
            } else {
                $sortHead.addClass('sorted-desc');
            }
            $table.find('td').removeClass('sorted').filter(':nth-child(' + (column + 1) + ')').addClass('sorted');
            $table.alternateRowColors().trigger('repaginate');
        });
    }
});

```

```

    });
  });
}
</script>

```

10.2 数据分页

与排序不同,数据分页多发生在服务器端,即通过与服务器端交互,由服务器控制显示的页数和分页数据。用户可以通过在查询字符串中向服务器发送信息来触发,如“test.asp?page=2;”也可以通过加载完整的页面,或者通过 Ajax 只提取一段表格数据来完成这个任务。

10.2.1 服务器端分页

通过服务器端分页的设计思路是:利用 SQL 字符串查询出需要的数据,然后根据一定的分页逻辑,即以记录集对象的分页属性,来确定每次从服务器端发送给客户端的记录数和逻辑页记录集在整个查询的记录集中的位置。

【示例 1】设计每页显示记录数为 2 条,整个记录集包含 14 条记录。当页面初次加载时显示前 2 条记录,单击“下一页”按钮,会向服务器端发送一个请求,并通过 Ajax 技术异步与服务器进行数据交互,然后在客户端显示出来。演示效果如图 10.3 所示。

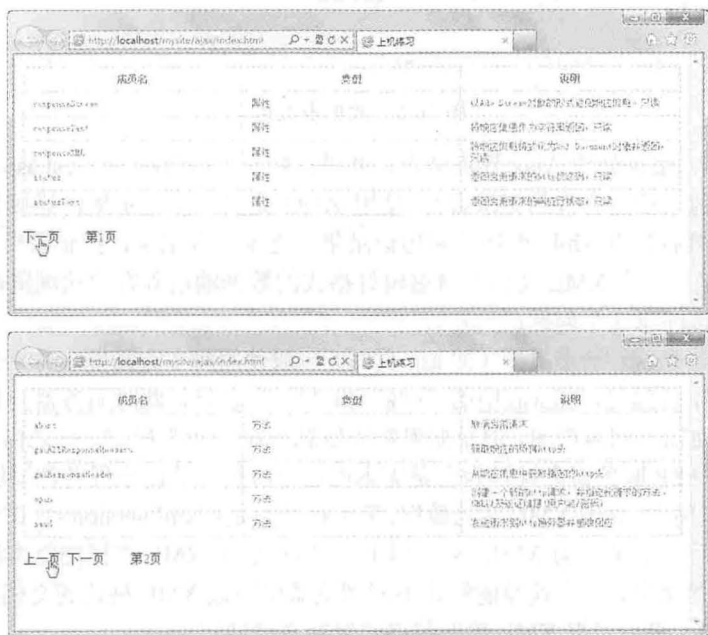


图 10.3 分页显示效果

(1) 设计一个简单的 Access 数据库,名称为 data.mdb,数据库中包含一个数据表,名称为 xmlhttp。数据表的结构如图 10.4 所示,数据表的数据如图 10.5 所示。xmlhttp 表中包含 4 个字段: id (自动编号数据类型、序列号,由数据库自动生成)、who (字符串数据类型,表示成员名称)、class (字符串数据类型,表示成员类型,如属性或方法)和 what (字符串数据类型,表示对成员说明)。

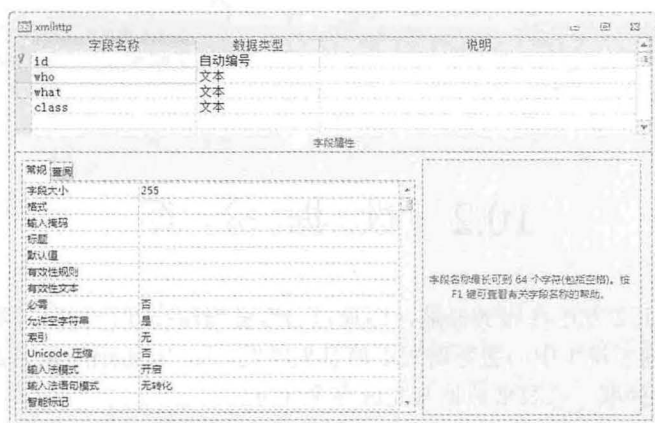


图 10.4 数据表结构



图 10.5 数据表数据

(2) 编写后台脚本, 用来处理 Ajax 异步请求, 并进行响应。后台脚本 (test.asp) 的设计思路是: 获取客户端传递过来的参数值 (指定查询的记录数), 使用 ADO 定义一个记录集, 连接到后台数据库, 并查询指定记录数的记录集。然后利用 while 循环体遍历记录集, 逐条读取记录, 把记录转换为 XML 格式数据。为此, 根据 XML 格式编写一个 XML 文档并将编辑好格式的数据输出到客户端浏览器。

在编写脚本时, 应该注意 3 个问题:

- ☑ 对于后台服务器来说, 动态页面 (如 asp 页面) 一般使用脚本方法输出数据到服务器, 可以把数据写在服务器脚本分隔符, 如 asp 中的 “<%” 和 “%>” 之外, 此时服务器会把它们看作是响应信息串, 原封不动地输出到客户端。但是如果响应信息写在 “<%” 和 “%>” 之内, 就必须包含在 write() 方法中, 否则 ASP 服务器会误以为它是脚本而进行解析, 从而出现各种语法错误。
- ☑ XML 文档格式问题。对于 XML 文档数据, 第 1 行必须是 <?xml version="1.0" encoding="gb2312"?>。该行命令表示输出的数据为 XML 格式文档, 同时标识了 XML 文档的版本和字符编码。为了能够兼容 IE 和 FF 等浏览器, 也就是能够让不同浏览器识别该 XML 格式的文档, 不要忘记为文档定义 XML 文本类型。最后根据 XML 数据的格式编写文档结构。

如同 HTML 文档一样, XML 文档外围也是一个根节点 (本例为 <data>), 节点内可以包含属性, 根节点内包含 <item> 子节点。如果说 <data> 根节点表示数据表, 则 <item> 子节点就表示一条记录, 而 <item> 子节点中包含的 3 个孙子节点 (<who>、<class> 和 <what>) 分别对应数据表中的 3 列数据。

- ☑ 关于 ADO 操作问题。ADO 是 ASP 服务器中的一个大型组件, 专门用来读写数据库, 它与 JavaScript 客户端的 DOM 外挂组件有点形似, 但是功能各异。ADO 与 DOM 组件一样, 不需要引用, 直接在包含的环境中使用即可。要使用这些组件, 先应该实例化指定对象, 然后调用对象的方法和属性来实现各种功能操作。

ADO 读写数据库一般包括 4 个操作步骤:

- ① 建立与指定数据库的连接。
- ② 定义记录集。
- ③ 利用定义的记录集从数据库中查询 SQL 语句中定义的数据。
- ④ 借助 ASP 脚本 (以 VBScript 脚本语言为主), 循环读取记录集中的记录并显示。

ASP 脚本文件 (test.asp) 的完整代码如下:

```
<?xml version="1.0" encoding="gb2312"?>
<%
'该输出命令曾经是前面示例中的一行后台代码, 该行代码表示定义输出字符编码, 但是 IE 会因此把文档误认为
HTML 文档, 从而拒绝以 XML 格式读取数据, 所以在这里特意请读者删除它
'Response.AddHeader "Content-Type", "text/html; charset=gb2312"
Response.ContentType = "text/xml" '定义 XML 文档文本类型, 否则 IE 浏览器将不识别
实例化数据库连接对象
set conn = Server.CreateObject("adodb.connection")
data = Server.mappath("data.mdb") '获取数据库的物理路径
conn.Open "driver={microsoft access driver (*.mdb)};&"&"dbq="&data '利用数据库连接对象打开数据库
coun=CInt(Request("coun")) '获取客户端传递过来的参数, 并转为数值, 以便进行运算
%>
<%
coun=CInt(Request("coun")) '获取客户端传递过来的记录集指针位置
if coun<1 then coun = 1 '如果记录集指针小于 1, 则设置为 1, 避免指针溢出
if coun>14 then coun = 14 '如果记录集指针大于 14, 则设置为 14, 避免指针溢出
%>
<% '定义并打开记录集
set rs = Server.CreateObject("adodb.recordset") '定义记录集对象
sql ="select * from xmlhttp" '定义 SQL 查询字符串
rs.CursorType=3 '设置指针类型为 3, 这样可以来回移动指针
rs.CursorLocation = 3 '设置记录集锁定类型为 3
rs.open sql,conn,2,1 '打开记录集
rs.AbsolutePosition = coun '把记录集的指针移到参数指定的位置
%>
<!-- 以下脚本和代码都是用来输出 XML 文档结构和数据信息的 -->
<data count="<%=coun%>" >
<!-- 输出根节点, 并定义一个属性, <%=coun%>表示 ASP 脚本输出的意思 -->
<%
n=0
while (not rs.eof) and (n<5) '循环读取记录集中当前指针开始的两条记录
%>
    <item id="<%=rs("id")%>"> '输出子节点 -->
        <who><%=trim(rs("who")) %></who><!-- 输出孙子节点 -->
        <class><%=trim(rs("class")) %></class><!-- 输出孙子节点 -->
        <what><%=trim(rs("what")) %></what><!-- 输出孙子节点 -->
    </item>
<%
n = n + 1 '递增循环次数
rs.movenext '向下移动记录集指针, 以读取下一条记录
wend
%>
</data> '输出根节点的结束节点 -->
```

在上面 ASP 输出语句中, “<%= %>” 表示一种快速输出方法, 它能够很自由地在文档中输出脚本变量信息。另外, “<%=trim(rs("who")) %>” 表示输出记录集中指定字段的值, trim() 函数表示清除左右两侧的

最后需要补充有关 ADO 数据库操作的两个知识点。

- ☑ 记录集当前指针：ADO 规定记录集中在同一时间只能够有一条记录可以读写，这条记录就是当前记录。为了便于控制这个当前记录，ADO 定义了一个指针（或称为标记），记录集指针时刻都会指着当前记录的位置。当需要读写下一条记录时，记录集指针随之移动位置。所以要想读写记录集中某一条记录，必须先把记录集的指针移到对应的位置，然后才能够操作，此时可以使用 Recordset 对象的 AbsolutePosition 进行读写控制。
- ☑ 记录集指针类型：记录集的指针有多种类型，这些类型可以通过 Recordset 对象的 CursorType 属性进行控制。取值为 0，表示记录集指针只能向前移动，不能向后移动；取值为 1，表示可以前后移动，但是其他用户的操作不会影响当前记录集，除了其他用户删除记录外；取值为 2，表示可以前后移动指针，其他用户的任何操作都会同时影响当前记录集结果；取值为 3，表示可以前后移动指针，但是对于其他用户的操作不受影响。

(3) 完成后台设计后, 再来设计前台结构和脚本。前台结构如下 (index.html):

```
<body onload="check();">  
<div id="info"></div>  
<p><span class="btn" id="up" onclick="check(1)">上 一 页</span> <span class="btn" id="down" onclick=  
"check(2)">下 一 页</span>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<span>第<span class="red" id="cur">1</span>页  
</span></p>  
</body>
```

在 body 中绑定异步处理函数，实现页面初始化显示“第 1 页”记录。然后在标题标签中嵌套一个 span，用来动态输出显示当前页数，在后面定义两个按钮（span 元素），绑定异步处理函数，分别设置传递值为 1 和 2，以告诉脚本当前按钮是往前还是往后翻页操作。

(4) 根据翻页按钮的操作来计算翻页后的记录集指针位置。由于已经知道每页显示记录数和总记录集数, 所以设计的代码就比较直观了。具体实现代码如下:

306

```

    }else {          //否则显示“下一页”按钮
        document.getElementById( "down" ).style.display = "inline";
    }
}
request.open( "GET", "test.asp?coun=" + coun, true );      //打开服务器连接，并传递指针位置值
request.onreadystatechange = updatePage;                  //绑定回调函数
request.send( null );                                     //发送请求
}

```

(5) 定义回调函数。该函数不需要传递参数，代码如下：

```

function updatePage(){
    var info = document.getElementById( "info" );
    if( request.readyState == 1 ){
        info.innerHTML = "<img src='../images/loading.gif' />， 连接中.....";
    }
    else if( request.readyState == 2 || request.readyState == 3 ){
        info.innerHTML = "<img src='../images/loading.gif' />， 读数据.....";
    }
    }else if( request.readyState == 4 ){
        if( request.status == 200 ){
            xml = request.responseXML;
            info.innerHTML = showXml( xml );
        }else
            alert( request.status );
    }
}
}

```

10.2.2 JavaScript 实现分页

JavaScript 实现分页的效果是一种客户端特效，它与服务器端分页有着本质不同。JavaScript 实现分页的数据实际上都已经存在于客户端，只是在视觉上进行了隐藏和显示处理，而服务器端分页只是分页响应数据给客户端。

下面介绍如何通过 JavaScript 对浏览器中已经存在的数据表格进行分页。现在从显示特定数据页开始，如仅显示数据表格中前 10 页数据（即第 1 页），实现代码如下：

```

$(function() {
    $('table.paginated').each(function() {
        var currentPage = 0;
        var numPerPage = 10;
        var $table = $(this);
        $table.find('tbody tr').show()
            .slice(0, currentPage * numPerPage)
            .hide()
            .end()
            .slice((currentPage + 1) * numPerPage)
            .hide()
            .end();
    });
}

```

首先，为分页表格绑定一个类标识（paginated），这样就可以在脚本中针对 \$('table.paginated') 进行处理。这里有两个控制变量：currentPage 指定当前显示的页，从 0 开始；numPerPage 指定每页要显示的数据行。

在 each() 参数中回调函数体内的 this 指向当前数据表格（table 元素），故需要使用 \$() 构造函数把它转换为 jQuery 对象。利用 tbody 元素作为标识符，把标题和数据行分离出来，使用 show() 显示所有数据行，然后调用 slice() 方法过滤出指定范围前的数据行，并把它们隐藏起来。为了统一操作对象，在调用 hide() 方法

后, 调用 `end()` 方法恢复最初操作的 jQuery 对象。以同样的方式, 隐藏特定范围后面的所有行。

为了方便用户选择分页, 还需要动态设置分页指示按钮。虽然可以使用超链接来实现分页指向功能, 但是这违反了 JavaScript 动态控制的原则, 反而让超链接的默认行为影响用户操作, 容易导致误操作。为此这里通过脚本形式动态创建几个 DOM 元素, 并通过数字标识分页向导。代码如下:

```
var numRows = $table.find('tbody tr').length;
var numPages = Math.ceil(numRows / numPerPage);
var $pager = $('<div class="pager"></div>');
for(var page = 0; page < numPages; page++) {
    $('<span class="page-number">' + (page + 1) + '</span>').appendTo($pager).addClass('clickable');
}
```

通过数据行数除以每页显示的行数, 即可得到分页的页数。如果得到的结果不是整数, 必须使用 `Math.ceil()` 方法向上舍入, 以确保显示最后一页。然后根据这个数字, 就可以为每个分页创建导航按钮, 并把这个新的导航按钮附加到表格前面。演示效果如图 10.6 所示。

ID	产品名称	标准成本	销售价格	单位数量	最小购买数量	类别
1	军帽	5.00	50.00	100个	10	军帽
2	军帽	4.00	40.00	100个	10	军帽
3	军帽	4.00	40.00	100个	10	军帽
4	军帽	12.00	40.00	100个	10	军帽
5	军帽	5.00	50.00	100个	10	军帽
6	军帽	20.00	40.00	100个	10	军帽
7	军帽	15.00	20.00	100个	10	军帽
8	军帽	12.00	20.00	100个	10	军帽
9	军帽	3.00	5.00	100个	10	军帽
10	军帽	10.00	45.00	100个	10	军帽

图 10.6 分页导航

在内部样式表中设计按钮的样式, 以方便用户操作。代码如下:

```
<style type="text/css">
.page-number { padding: 0.2em 0.5em; border: 1px solid #fff; cursor:pointer; display:inline-block; }
.active { background: #ccf; border: 1px solid #006; }
</style>
```

其中样式类 `active` 表示当前激活的分页按钮。此时的按钮演示效果如图 10.7 所示。

ID	产品名称	标准成本	销售价格	单位数量	最小购买数量	类别
27	军帽	4.00	12.00	100个	10	军帽
63	军帽	4.00	20.00	100个	10	军帽
95	军帽	10.00	20.00	100个	10	军帽
12	军帽	3.00	5.00	100个	10	军帽
74	军帽	5.00	45.00	100个	10	军帽
77	军帽	3.00	18.00	100个	10	军帽
90	军帽	2.00	10.00	100个	10	军帽
91	军帽	4.00	20.00	100个	10	军帽

图 10.7 分页导航按钮

要实现分页导航功能, 需要实现动态更新 `currentPage` 变量, 同时运行上面的分页脚本。可以把上面的代码封装到一个函数中, 每当单击导航按钮时, 更新 `currentPage` 变量, 并调用该函数。

在循环体中为每个按钮绑定 `click` 事件处理函数, 由于创建了一个闭包体, 闭包体内引用了外部的 `currentPage` 变量, 当每个循环改变时, 该变量的值就会发生变化, 新的值会影响到每个按钮上绑定的闭包体 (`click` 事件处理函数)。为了解决这个问题, 用户不妨利用 jQuery 的事件绑定方法中的一个高级特性, 可

以在绑定处理函数时为它添加一组数据，这组数据在最终调用时仍然有效。代码如下：

```
for(var page = 0; page < numPages; page++) {
    $('<span class="page-number">' + (page + 1) + '</span>').bind('click', {
        'newPage' : page
    }, function(event) {
        currentPage = event.data['newPage'];
        //省略分页函数
    })
}
```

在 for 循环体内，为每个导航按钮绑定一个 click 事件处理函数，并通过事件对象的 data 属性为其传递动态的当前页数值，这样 click 事件处理函数所形成的闭包体就不再直接引用外部的变量，而是通过事件对象的属性来获取当前页信息，从而避免了闭包的缺陷。

为了突出显示当前页，可以在 click 事件中添加如下一行代码，为当前导航按钮添加一个样式类，以激活当前按钮，方便用户浏览。

```
for(var page = 0; page < numPages; page++) {
    $('<span class="page-number">' + (page + 1) + '</span>').bind('click', {
        'newPage' : page
    }, function(event) {
        currentPage = event.data['newPage'];
        //省略分页函数
        $(this).addClass('active').siblings().removeClass('active');
    }).appendTo($pager).addClass('clickable');
}
```

最后，还需要把这个分页导航插入到网页中，同时把分页函数绑定到 repaginate 事件处理函数中，这样就可以通过 \$table.trigger('repaginate') 方法快速调用。整个数据表格分页功能的完整脚本代码如下：

```
$(function() {
    $('table.paginated').each(function() {
        var currentPage = 0;
        var numPerPage = 10;
        var $table = $(this);
        $table.bind('repaginate', function() {
            $table.find('tbody tr').show().slice(0, currentPage * numPerPage).hide().end().slice((currentPage + 1) * numPerPage).hide().end();
        });
        var numRows = $table.find('tbody tr').length;
        var numPages = Math.ceil(numRows / numPerPage);
        var $pager = $('<div class="pager"></div>');
        for(var page = 0; page < numPages; page++) {
            $('<span class="page-number">' + (page + 1) + '</span>').bind('click', {
                'newPage' : page
            }, function(event) {
                currentPage = event.data['newPage'];
                $table.trigger('repaginate');
                $(this).addClass('active').siblings().removeClass('active');
            }).appendTo($pager).addClass('clickable');
        }
        $pager.find('span.page-number:first').addClass('active');
        $pager.insertBefore($table);
        $table.trigger('repaginate');
    });
});
```


最终演示效果如图 10.8 所示。

ID	产品名称	标准成本	特价价格	单位重量	最小订购数量	类别
20	桂花糕	20.00	20.00	每箱20盒	10	糕点
21	花生	15.00	15.00	每箱20包	5	坚果食品
24	香蕉	10.00	20.00	每箱20箱	15	饮料
40	苹果	0.00	20.00	每箱20斤	20	水果类
41	橙子	0.00	20.00	每箱20斤	10	水果
42	猕猴桃	10.00	20.00	每箱2箱	25	饮料
43	李子梅	5.00	15.00	每箱2箱	25	点心
51	猪肉干	15.00	40.00	每箱2箱	10	干果和坚果
52	三合一果冻	10.00	30.00	每箱2箱	25	饮料/果汁
53	甜米	2.00	10.00	每箱20斤	20	意大利面等

图 10.8 数据表格分页导航

10.3 数据过滤

数据过滤也是表格设计中的一项基本功能。在大量数据显示的表格中，如果允许用户根据需要仅显示特定内容的数据行，能够极大地提高表格的可用性。

10.3.1 快速过滤数据

使用 JavaScript 实现表格数据过滤的基本功能很简单，即通过检索用户输入的关键字，把匹配的行隐藏或者显示出来，没有被匹配的行显示或者隐藏起来。代码如下：

```
var elems = $('table.filter').find("tbody > tr")
elems.each(function() {
    var elem = jQuery(this);
    jQuery.uiTableFilter.has_words(getText(elem), words, false) ? matches(elem) : noMatch(elem);
});
```

在上面几行代码中，首先找到要检索的数据行，这里主要是根据 table 和过滤类确定要过滤的表格，并根据 tbody 元素确定检索的数据行。遍历数据行，使用用户输入的过滤关键字与每行单元格数据进行匹配。如果返回 true，则执行显示操作，否则执行隐藏操作。其中 getText() 是一个内部函数，用户获取指定行中单元格包含的文本。代码如下：

```
var getText = function(elem) {
    return elem.text()
}
```

has_words() 是数据过滤插件的一个工具函数，该函数主要检测用户输入关键字与数据行文本是否匹配。代码如下：

```
jQuery.uiTableFilter.has_words = function(str, words, caseSensitive) {
    var text = caseSensitive ? str : str.toLowerCase();
    for(var i = 0; i < words.length; i++) {
        if(text.indexOf(words[i]) === -1)
            return false;
    }
    return true;
}
```

该工具函数首先根据一个 caseSensitive 参数，确定是否把数据行文本执行小写转换，然后遍历用户输入的关键字数组，执行匹配计算。如果不匹配，则返回 false，否则返回 true。

matches()和 noMatch()是两个简单的显示和隐藏数据行内部函数。代码如下:

```
var matches = function(elem) {
    elem.show()
}
var noMatch = function(elem) {
    elem.hide();
    new_hidden = true
}
```

10.3.2 处理多关键字匹配

如果当用户输入多个关键字,则数据过滤器应该允许对多个关键字的协同处理。首先可以通过 JavaScript 的 split()方法把用户输入的短语以空格符为分隔符劈开,然后转换为数组。代码如下:

```
var words = phrase.toLowerCase().split(" ");
```

执行数据过滤的事件一般设置为键盘松开时触发,当用户在搜索框中输入关键字时,会即时触发并更新过滤数据。为了避免因为用户输入空格键而触发重复的数据过滤操作,则设置一个检测条件。当输入字符之后,去除最后一个空格符,如果等于上次输入的字符,则说明当前输入的是空格,可以不做重复检测,这样就能够提高过滤效率。

```
if((words.size > 1) && (phrase.substr(0, phrase_length - 1) === this.last_phrase)) {
    if(phrase[-1] === " ") {
        this.last_phrase = phrase;
        return false;
    }
    var words = words[-1];
    //获取可见数据行
    var elems = jq.find("tbody > tr:visible")
}
```

在上面代码中将根据用户输入的多个关键字进行处理,关键字之间通过空格符进行分隔,同时当输入最新关键字时,代码只处理可视的数据行,对于已经隐藏的数据行将忽略不计。

10.3.3 处理特定列过滤

在过滤器函数中包含一个列参数,允许用户仅就特定列数据进行过滤。代码如下:

```
f(column) {
    var index = null;
    jq.find("thead > tr:last > th").each(function(i) {
        if($(this).text() == column) {
            index = i;
            return false;
        }
    });
    if(index == null)
        throw ("given column: " + column + " not found")
    getText = function(elem) {
        return jQuery(elem.find(("td:eq(" + index + ")"))).text()
    }
}
```

参数 column 表示列标题,代码首先遍历数据表格的列标题,匹配参数 column 列的下标位置,然后利用该下标值,重写 getText()内部函数,则执行匹配操作时,仅就该下标列的文本进行匹配检测。

10.3.4 合成数据过滤器

到目前为止，数据过滤的功能基本实现，下面就可以把这个数据过滤器的各个部分功能代码组合在一起，放置在 jQuery 工具函数中。代码如下：

```
jQuery.ui.TableFilter = function(jq, phrase, column, ifHidden) {
    var new_hidden = false;
    if(this.last_phrase === phrase)
        return false;
    var phrase_length = phrase.length;
    var words = phrase.toLowerCase().split(" ");
    var matches = function(elem) {
        elem.show()
    }
    var noMatch = function(elem) {
        elem.hide();
        new_hidden = true
    }
    var getText = function(elem) {
        return elem.text()
    }
    if(column) {
        var index = null;
        jq.find("thead > tr:last > th").each(function(i) {
            if($(this).text() == column) {
                index = i;
                return false;
            }
        });
        if(index == null)
            throw ("given column: " + column + " not found")
        getText = function(elem) {
            return jQuery(elem.find(("td:eq(" + index + ")"))).text()
        }
    }
    if((words.size > 1) && (phrase.substr(0, phrase_length - 1) === this.last_phrase)) {
        if(phrase[-1] === " ") {
            this.last_phrase = phrase;
            return false;
        }
        var words = words[-1];
        var elems = jq.find("tbody > tr:visible")
    } else {
        new_hidden = true;
        var elems = jq.find("tbody > tr")
    }
    elems.each(function() {
        var elem = jQuery(this);
        jQuery.ui.TableFilter.has_words(getText(elem), words, false) ? matches(elem) : noMatch(elem);
    });
    last_phrase = phrase;
}
```

```

    if(ifHidden && new_hidden)
        ifHidden();
    return jq;
};
jQuery.uiTableFilter.last_phrase = ""
jQuery.uiTableFilter.has_words = function(str, words, caseSensitive) {
    var text = caseSensitive ? str : str.toLowerCase();
    for(var i = 0; i < words.length; i++) {
        if(text.indexOf(words[i]) === -1)
            return false;
    }
    return true;
}

```

完成工具函数的设计，就可以在文档中设计一个文本输入框和一个需要过滤的数据表格。代码如下：

```

<form>数据过滤: <input name="filter" id="filter" value="" maxlength="30" size="30" type="text"></form>
<table class="filter">
  <thead>
    <tr>
      <th>ID</th>
      <th>产品名称</th>
      <th>标准成本</th>
      <th>列出价格</th>
      <th>单位数量</th>
      <th>最小再订购数量</th>
      <th>类别</th>
    </tr>
  </thead>
  <tbody>
    ...
  </tbody>
</table>

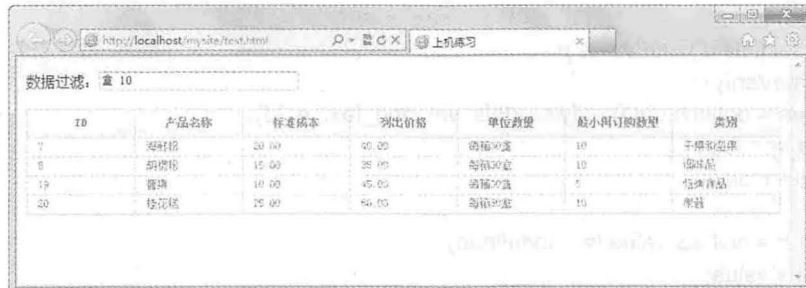
```

在文档头部的<script>标签中为文本框绑定过滤器函数，设计当用户在文本框中输入字符后，键盘按钮松开时，触发该工具函数\$.uiTableFilter()，并把数据表格和当前文本框的输入值传递给它。最后示例演示效果如图 10.9 所示。

```

$(function() {
    var theTable = $('table.filter')
    $("#filter").keyup(function() {
        $.uiTableFilter(theTable, this.value);
    })
})

```



ID	产品名称	标准成本	列出价格	单位数量	最小再订购数量	类别
7	海鲜松	20.00	40.00	每桶20盒	10	干燥和凉菜
8	鸡蛋松	15.00	30.00	每桶20盒	10	凉味品
19	露味	10.00	45.00	每桶20盒	5	佐味食品
20	松花松	25.00	50.00	每桶20盒	10	凉菜

图 10.9 数据表格过滤

10.4 数据编辑

表格数据可直接编辑是表格易用性最重要的特征之一。当用户在浏览大容量表格数据时，能够即点即改，提高数据加工效率，避免表格与表单页面来回切换执行相关编辑操作的麻烦。表格数据编辑功能主要体现在数据编辑、验证和存储。本节将重点讲解如何实现数据表格的直接编辑，对于编辑后的验证和存储问题留给读者自己尝试解决。

10.4.1 快速编辑数据

当用户单击单元格时，单元格显示为可编辑状态，数据可以删除、修改或者增删。要实现这样的功能，可以在鼠标单击事件处理函数中进行这样的设计：获取单元格数据，动态创建一个文本框，文本框的值为单元格的数据，然后把该文本框嵌入到单元格中，并清除单元格中的原始数据。代码如下：

```
var orig_text = td.text();
var w = td.width();
var h = td.height();
td.css({
    width : w + "px",
    height : h + "px",
    padding : "0",
    margin : "0"
});
td.html('<form name="td-editor" action="javascript:void(0);">' + '<input type="text" name="td_edit" value="' +
td.text() + '" + ' style="margin:0px;padding:0px;border:0px;width: ' + w + 'px;height:' + h + 'px;">' +
'</input></form>');
```

在上述代码中，首先保存单元格原始数据，获取单元格的高度和宽度，再显式定义单元格的高、宽并清除空隙，避免清除原始数据后，单元格大小发生变化。然后使用 `html()` 方法在单元格中绑定一个 `<form>` 和 `<input>` 标签，在标签内部通过样式属性定义输入文本框的大小与单元格大小一致，并清除间距。

当数据编辑完成之后，需要把文本框清除掉，并使用编辑后的值更新单元格的原始值。代码如下：

```
function restore(e) {
    var val = td.find(':text').attr('value')
    td.html("");
    td.text(val);
}
```

在执行恢复单元格数据过程中，可以预留两个接口函数，以便用户通过参数传递数据验证函数，以及操作之后的回调函数。代码如下：

```
function restore(e) {
    var val = td.find(':text').attr('value')
    if(options.dataVerify) {
        var value = options.dataVerify.call(this, val, orig_text, e, td);
        if(value === false) {
            return false;
        }
        if(value !== null && value !== undefined)
            val = value;
    }
    td.html("");
```

```

td.text(val);
if(options.editDone)
    options.editDone(val, orig_text, e, td)
bind_mouse_down(td_edit_wrapper);
}

```

options.dataVerify 作为一个参数，为数据验证提供接口，只有当验证函数返回值为 true，才允许编辑操作，否则禁止编辑并返回。

options.editDone 也是一个参数，为数据编辑完成后的回调函数，在回调函数中可以执行一些附加的任务或者功能。

当完成数据编辑之后，需要调用 restore() 函数，把数据恢复为表格数据，并清除表单元素。此时可以在添加的表单元素中绑定提交、鼠标按下、失去焦点等事件中绑定 restore() 函数。代码如下：

```

td.html('<form name="td-editor" action="javascript:void(0);">' + '<input type="text" name="td_edit" value="" +
td.text() + "" + ' style="margin:0px;padding:0px;border:0px;width: ' + w + 'px;height:' + h + 'px;">' + '</input>' +
'</form>').find('form').submit(restore).mousedown(restore).blur(restore);

```

10.4.2 完善数据编辑功能

在该插件函数中，定义两个事件处理函数：bind_mouse_down() 函数负责绑定鼠标按下事件处理函数，unbind() 函数将注销该事件。具体用法如下：

```

function unbind() {
    return jq.find(options.find).die('mousedown.uiTableEdit');
}
function bind_mouse_down(mouseDn) {
    unbind().live('mousedown.uiTableEdit', mouseDn);
}

```

考虑到表格数据编辑的扩展性，这里留给用户一个接口，允许自定义单元格编辑个性定制功能。如果用户提供了 mouseDown 这个参数函数，将调用该函数，并检测返回值。如果为 false，则结束程序；否则调用单元格编辑函数，执行编辑操作。代码如下：

```

var td_edit_wrapper = !options.mouseDown ? td_edit : function() {
    if(options.mouseDown.apply(this, arguments) == false)
        return false;
    td_edit.apply(this, arguments);
};

```

由于 Firefox 在获取焦点时存在一个 Bug，该浏览器在稍稍延迟之后，才能够获取焦点，为此需要定义一个获取焦点函数，并在延迟 50 毫秒之后调用这个延迟函数。代码如下：

```

function focus_text() {
    td.find('input:text').get(0).focus()
}
setTimeout(focus_text, 50);

```

如果用户按下 Esc 键，允许用户退出表格数据编辑状态。添加如下函数，并调用该函数的事件处理函数代码：

```

function checkEscape(e) {
    if(e.keyCode === 27) {
        td.html("");
        td.text(orig_text);
        bind_mouse_down(td_edit_wrapper);
    }
}

```

```

td.html('<form name="td-editor" action="javascript:void(0);">' + '<input type="text" name="td_edit" value="" +

```

```
td.text() + "" + ' style="margin:0px;padding:0px;border:0px;width: ' + w + 'px;height:' + h + 'px;">' +
'</input></form>').find('form').submit(restore).mousedown(restore).blur(restore).keypress(checkEscape);
```

在该插件函数起始部分, 应该对参数对象进行处理, 并根据 options.off 参数值, 确定是否执行编辑操作。

代码如下:

```
options = options || {}
options.find = options.find || 'tbody > tr > td'
if(options.off) {
    unbind().find('form').each(function() {
        var f = $(this);
        f.parents("td:first").text(f.find(':text').attr('value'));
        f.remove();
    });
    return jq;
}
```

到目前为止, 数据编辑的功能基本实现, 下面就可以把这个数据编辑器的各个部分功能代码组合在一起, 放置在 jQuery 工具函数中。代码如下:

```
jQuery.uiTableEdit = function(jq, options) {
    function unbind() {
        return jq.find(options.find).die('mousedown.uiTableEdit');
    }
    options = options || {}
    options.find = options.find || 'tbody > tr > td'
    if(options.off) {
        unbind().find('form').each(function() {
            var f = $(this);
            f.parents("td:first").text(f.find(':text').attr('value'));
            f.remove();
        });
        return jq;
    }
    function bind_mouse_down(mouseDn) {
        unbind().live('mousedown.uiTableEdit', mouseDn);
    }
    function td_edit() {
        var td = jQuery(this);
        function restore(e) {
            var val = td.find(':text').attr('value')
            if(options.dataVerify) {
                var value = options.dataVerify.call(this, val, orig_text, e, td);
                if(value === false) {
                    return false;
                }
            }
            if(value !== null && value !== undefined)
                val = value;
        }
        td.html("");
        td.text(val);
        if(options.editDone)
            options.editDone(val, orig_text, e, td)
        bind_mouse_down(td_edit_wrapper);
    }
    function checkEscape(e) {
        if(e.keyCode === 27) {
```

```

        td.html("");
        td.text(orig_text);
        bind_mouse_down(td_edit_wrapper);
    }
}
var orig_text = td.text();
var w = td.width();
var h = td.height();
td.css({
    width : w + "px",
    height : h + "px",
    padding : "0",
    margin : "0"
});
td.html('<form name="td-editor" action="javascript:void(0);">' + '<input type="text" name="td_edit"
value="' + td.text() + '" style="margin:0px;padding:0px;border:0px;width: ' + w + 'px;height:' + h + 'px;">' +
'</input></form>').find('form').submit(restore).mousedown(restore).blur(restore).keypress(checkEscape);
function focus_text() {
    td.find('input:text').get(0).focus()
}
setTimeout(focus_text, 50);
bind_mouse_down(restore);
}
var td_edit_wrapper = !options.mouseDown ? td_edit : function() {
    if(options.mouseDown.apply(this, arguments) == false)
        return false;
    td_edit.apply(this, arguments);
};
bind_mouse_down(td_edit_wrapper);
return jq;
}

```

在文档头部的<script>标签中为文本框绑定编辑器函数，最后示例演示效果如图 10.10 示。

```

$(function() {
    var theTable = $('#tableEdit')
    $.uiTableEdit( theTable )
})

```

ID	产品名称	标准成本	预计价格	单位数量	最小订购数量	类别
1	草莓汁	5.00	20.00	10箱 x 20包	10	饮料
2	葡萄汁	4.00	20.00	每箱10瓶	25	饮料品
3	苹果汁	3.00	25.00	每箱10瓶	10	饮料品
4	香蕉	11.00	40.00	每箱12瓶	10	饮料品
5	葡萄	5.00	20.00	每箱12瓶	25	葡萄
6	荔枝	20.00	40.00	每箱10瓶	10	干果和坚果
7	猕猴桃	15.00	25.00	每箱10瓶	10	饮料品
8	李子	11.00	20.00	每箱12瓶	10	干果和坚果
9	猪肉	2.00	9.00	每箱100斤	10	肉类和海鲜类
10	猪肉	234.00	45.00	每箱10瓶	5	肉类食品
11	桂花糕	35.00	50.00	每箱20包	10	糕点
12	荔枝	15.00	25.00	每箱10包	5	饮料食品
13	咖啡	10.00	30.00	每箱10瓶	15	饮料
14	茶叶	8.00	35.00	每箱10斤	20	饮料品
15	橙子	6.00	30.00	每箱10斤	10	茶
16	猕猴桃	10.00	20.00	每箱10瓶	25	饮料
17	玉米片	5.00	15.00	每箱10包	25	点心
18	猪肉干	15.00	40.00	每箱10包	10	干果和坚果
19	三文鱼片	12.00	30.00	每箱10包	25	肉类/鱼类
20	白米	5.00	10.00	每箱10斤	20	粮食和谷物
21	大米	5.00	10.00	每箱10斤	20	粮食和谷物

图 10.10 数据表格编辑

第 11 章

表单开发

( 视频讲解：2 小时 9 分钟)

最初的 Web 设计只是简单的论文展示，随着浏览者与作者沟通需求的日益强大，<form>标签由此诞生。

尽管当时并没有人意识到电子商务的诞生有可能会对人类生活产生巨大的影响，但是却没有谁能阻挡其前进的脚步。表单让网页发布者和网页使用者之间的对话成为了可能。表单不仅强大，还是一个有用的概念构想，因此今天几乎没有一个网站不使用它，有的网站甚至使用了多个表单。

从注册表单到联系信息表，从商业领域到政府部门，网页表单无处不在。成功的表单设计不仅能提高用户满意度，还能收集更加精确的数据并降低维护费用；而失败的表单设计不仅会收集到与需求相悖的冗余信息，还极有可能导致潜在消费者的流失。设计具有高可用性的表单绝非易事。本章将通过实例讲解如何设计出具有高可用性的优质网页表单。

11.1 设计可用性表单

表单在网页交互中扮演着非常重要的角色，它是任何网站与用户进行沟通的桥梁，客户端与服务器必须依靠表单才能实现数据交互。由于我们早已对日常生活中的表单习以为常，所以对于下面涉及的问题应努力避免：

- ☒ 所提问题让人不知如何回答的表单。
- ☒ 所提问题要求多项选择，但却没有提供所需答案的表单。
- ☒ 要求填写过多信息，或询问涉及回答者隐私信息的表单。
- ☒ 含有大量语义不明说明文字的表单。

在日常的谈话中，可以通过提问和语义解释来解决这些问题。但由于表单遵循的模式是：设计者告知表单所需的提问方式，表单精确地以这种方式向回答者提问，因此一个考虑周到的表单设计必须意思表达足够清楚，而且完全无须加以说明。

11.1.1 设计表单结构

当在网站中使用 jQuery 时，应该提醒自己，如果用户禁用了 JavaScript，那么页面看起来会是什么样的，功能是否健全，但是这并不否定使用 JavaScript 改善表单设计的倾向，渐进增强的设计原则：除努力为大部分用户提供额外功能外，还应该照顾全体用户的基本需求。

【示例 1】 下面创建一个表单，用来与用户建立联系，通过对它的外观和行为做渐进性增强演示，帮

助读者直观认识表单设计的可用性一般方法。具体代码如下，演示效果如图 11.1 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript"></script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
<form id="contact" action="index.html" method="get">
  <fieldset>
    <legend>个人信息</legend>
    <ol>
      <li>
        <label for="name">姓名</label>
        <input class="required" type="text" name="name" id="name" />
        <span>(必填)</span></li>
      <li>
        <label for="email">邮箱</label>
        <input class="required" type="text" name="email" id="email" />
        <span>(必填)</span></li>
      <li>如何保持联系? (至少选择一种)
        <ul>
          <li>
            <label>
              <input type="checkbox" name="by-contact-type" value="E-mail" id="by-email" />
              Email</label>
              <input class="conditional" type="text" name="email" id="email" />
              <span>(当勾选前面复选框后, 则必须填写 Email 信息)</span></li>
          <li>
            <label>
              <input type="checkbox" name="by-contact-type" value="Phone" id="by-phone" />
              电话</label>
              <input class="conditional" type="text" name="phone" id="phone" />
              <span>(当勾选前面复选框后, 则必须填写电话号码)</span></li>
          <li>
            <label>
              <input type="checkbox" name="by-contact-type" value="QQ" id="by-qq" />
              QQ</label>
              <input class="conditional" type="text" name="qq" id="qq" />
              <span>(当勾选前面复选框后, 则必须填写 QQ 号码)</span></li>
        </ul>
      </li>
    </ol>
  </fieldset>
</form>
</body>
</html>
```

上面代码中每个表单元素都包含在一个列表项()中, 所有元素都包含在一个有序列表()中, 而复选框以及对应的文本字段包含在一个嵌套的无序列表()中。而且, 这里使用<label>标签标出每个

字段的名称。对于文本字段，<label>标签放在<input>标签前面；而对于复选框，则<label>标签包含<input>标签。

图 11.1 设计联系表单

虽然在表单中可以添加各种辅助说明文字，以帮助或者引导用户填写表单中每个字段，但是还必须对能够使用的表单做进一步的改进。具体将从以下 3 个方面来增强表单的可用性：

- ☑ 修改 DOM，以便灵活地为<legend>元素应用样式。
- ☑ 把必填的字段提示信息改为星号，把特殊字段（即只需要勾选对应复选框时才能够填写的提示信息）修改为双星号。将这两个必填字段的标签修改为粗体字，同时在表单前面添加星号和双星号注释文字。
- ☑ 在页面加载时隐藏每个复选框对应的文本输入框，当用户选择或者取消复选框时能够动态切换这些文本框，让它们显示或者隐藏。

11.1.2 设计表单图标

<legend>标签用来表示标识图标，但是该标签在不同浏览器中的解析效果存在很大的差异性，使用 CSS 很难添加样式控制它，包括定位和浏览器兼容性问题。如果使用有意义的、结构良好的页面元素，那么<legend>标签适合作为表单分组的标题标签。

现在，设计通过 JavaScript 把页面中的每个<legend>标签移出，换成标题标签，即便个别用户禁用了 JavaScript，用户仍然能够根据语义化结构正常使用表单。

```
$(function() {
    $('fieldset').each(function(index) {
        var heading = $('legend', this).remove().text();
    });
});
```

使用 each() 方法遍历文档中所有的<legend>标签，使用 text() 方法获取该标签包含的文本，然后把<legend>标签移出文档。由于文档中包含多个表单，每个表单可能包含多个<legend>标签，因此简单使用 jQuery 的隐式迭代机制。同时要注意，由于每次迭代一个<fieldset>标签都会设置一个变量 heading，故需要使用 this 关键字限制匹配的范围，以确保每次只取得一个<legend>标签中的文本。

然后，创建 h3 元素，并把它插入到每个<fieldset>标签的开始位置，同时把保存到临时变量 heading 中的标题信息放入其中。代码如下，演示效果如图 11.2 所示。

```
$(function() {
    $('fieldset').each(function(index) {
        var heading = $('legend', this).remove().text();
        $('<h3></h3>')
        .text(heading)
        .prependTo(this);
    });
});
```

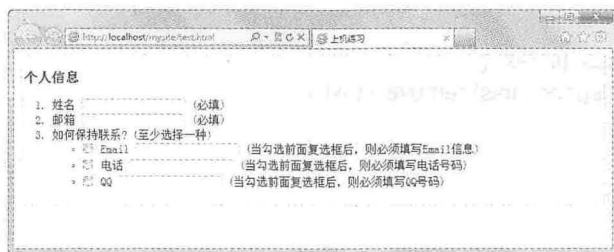


图 11.2 设计表单分组标题

11.1.3 设计提示信息

为了增加对必填字段的控制, 在设计表单结构时, 为这些字段添加 `required` 类, 通过这个样式类统一控制必填字段样式和行为。对于每种联系方式的输入文本框都有一个 `conditional` 类, 通过该类也可以对这些文本框进行控制。代码如下:

```
$(function() {
    //设置必填提示信息
    var requiredFlag = '*';
    var requiredKey = $('input.required:first').next('span').text();
    requiredKey = requiredFlag + requiredKey.replace(/\((.+)\)/, "$1");
    //设置必写提示信息
    var conditionalFlag = '**';
    var conditionalKey = $('input.conditional:first').next('span').text();
    conditionalKey = conditionalFlag + conditionalKey.replace(/\((.+)\)/, "$1");
    //附加信息
    $('form :input').filter('.required')
        .next('span').text(requiredFlag).end()
        .prev('label').addClass('req-label');
    $('form :input').filter('.conditional')
        .next('span').text(conditionalFlag);
})
```

在上面代码中, 先设置两个变量, 分别用来存储对应的提示星号, 并利用它们组合新的提示信息。由于星号很难吸引用户的注意力, 还应该为它们添加加粗样式, 即通过 `prev()` 方法获取前面的 `span` 标签, 并为其绑定一个样式类 `req-label`, 并为 `req-label` 样式类声明 `req-label { font-weight:bold;}`。

为了方便选择 `<label>` 标签, 在上面代码行中调用 `end()` 方法恢复上一次选择器所匹配的 jQuery 对象, 即从 `next('span')` 匹配的 `span` 元素返回到上一步的 `filter('.required')` 匹配的 `input` 文本框, 只有这样 `prev('label')` 才能够找到文本框前面的 `span` 元素。在生成保存的提示信息之前, 还应该把原始提示信息保存到变量中, 并通过正则表达式去掉前后的括号。演示效果如图 11.3 所示。

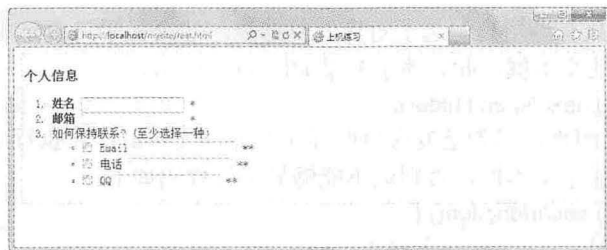


图 11.3 设计必填信息

最后, 把原始提示信息和标记符号一同放到表单的上面, 以便进行注释。代码如下:


```
$(function() {
    $('fieldset').each(function(index) {
        var heading = $('legend', this).remove().text();
        $('<h3></h3>')
            .text(heading)
            .prependTo(this);
    });
    var requiredFlag = ' * ';
    var requiredKey = $('input.required:first').next('span').text();
    requiredKey = requiredFlag + requiredKey.replace(/^(.+)\$/, "$1");
    var conditionalFlag = ' ** ';
    var conditionalKey = $('input.conditional:first').next('span').text();
    conditionalKey = conditionalFlag + conditionalKey.replace(/^(.+)\$/, "$1");
    $('form :input').filter('.required')
        .next('span').text(requiredFlag).end()
        .prev('label').addClass('req-label');
    $('form :input').filter('.conditional')
        .next('span').text(conditionalFlag);
    //添加注释信息
    $('<p></p>')
        .addClass('field-keys')
        .append(requiredKey + '<br />')
        .append(conditionalKey)
        .insertBefore("#contact");
})
```

在上面代码中, 首先创建一个 p 元素, 为该标签添加 field-keys 样式类, 将 requiredKey 和 conditionalKey 变量存储的信息附加到该标签中, 最后将该段落标签添加到联系表单的前面。演示效果如图 11.4 所示。

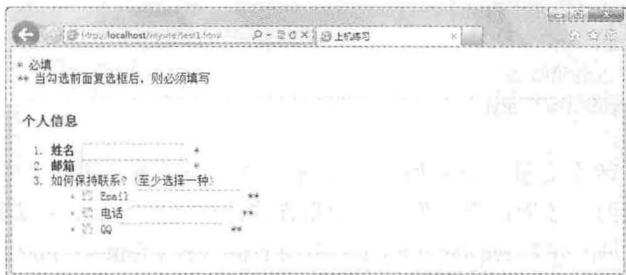


图 11.4 添加注释信息

11.1.4 设计条件字段

下面设计条件字段, 即只有当用户勾选了对应的复选框, 才会显示该复选框后面的文本框, 要求输入联系信息。首先, 隐藏所有的文本框, 此时演示效果如图 11.5 所示。

```
$('input.conditional').hide().next('span').hide();
```

为复选框添加一个 click 事件, 当勾选复选框时显示对应的文本框, 在执行过程中还应该检测复选框是否被选中。如果被选中, 则显示文本框, 否则就不能够显示。代码如下:

```
$('input.conditional').hide().each(function() {
    var $thisInput = $(this);
    var $thisFlag = $thisInput.next('span').hide();
    $thisInput.prev('label').find(':checkbox').click(function() {
        if (this.checked) {
```

```

    $thisInput.show().addClass('required');
    $thisFlag.show();
    $(this).parent('label').addClass('req-label');
  } else {
    $thisInput.hide().removeClass('required').blur();
    $thisFlag.hide();
    $(this).parent('label').removeClass('req-label');
  }
});
});

```

在上面代码中, 先保存当前文本输入字段和当前标记的变量。当用户单击复选框时, 需要检查复选框是否被选中。如果选中, 则显示文本框, 显示提示标记, 并为父元素<label>标签添加 req-label 样式类, 加粗显示标签文本。一般在检测复选框时, 可以通过在 each() 方法的回调函数中使用 this 关键字, 可直接访问当前 DOM 节点。如果不能够访问 DOM 节点, 则可以使用 \$(selector).is(':checked') 来代替, 因为 is() 方法返回值为布尔值。如果复选框被取消选中, 则应该隐藏文本框字段, 并清除父元素的 req-label 样式类。演示效果如图 11.6 所示。

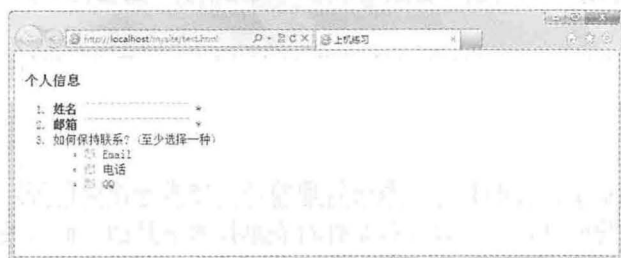


图 11.5 隐藏文本字段



图 11.6 显示条件文本字段

最后, 使用 CSS 在内部样式表中定义简单的样式, 适当美化联系表单。演示效果如图 11.7 所示。

```

<style type="text/css">
.req-label { font-weight:bold; }
h3 { background:#3CF; margin:0; padding:0.3em 0.5em; }
ul, ol { list-style-type:none; padding:0.5em; margin:0; }
ul { margin-left:1.5em; }
li { margin:4px; }
#contact { position:relative; }
p { position:absolute; right:1em; top:2em; background:#CFC; padding:1em; }
</style>

```

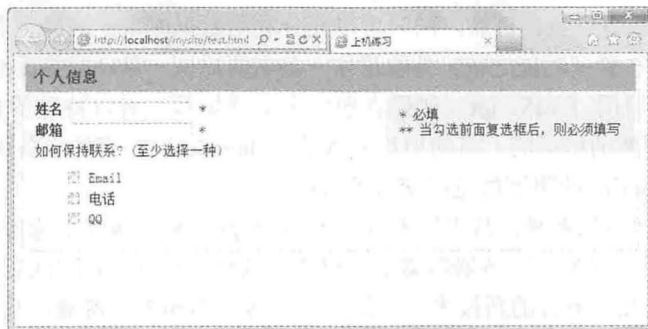


图 11.7 美化后的联系表单样式

11.2 表 单 验 证

验证是网站开发的一项核心功能，它犹如表单的防火墙，时刻保护交互数据的合法和安全，保证站点能够正确、准确地运行。随着互联网上相互交往变得愈加丰富和复杂，如何保障信息安全，保证数据完整、准确地传递是 Web 应用开发中必须要考虑的问题。

11.2.1 验证服务概述

任何 Web 应用程序都离不开表单，表单作为客户端向服务器端提交信息的载体担当着重要的角色。验证服务主要是针对表单的，不管是关于表单信息输入的验证，还是有关用户身份的验证，都与表单存在某种联系。实际上，Web 应用程序中的验证服务包含的内容是比较广泛的。

为了确保用户输入的信息合法，表单验证是表单设计中的一个重要环节。如果读者经常在网页中输入信息，也会遇到各种表单所提示的错误信息。这些信息提示如何输入合法的信息，避免浪费不必要的时间。

表单验证的过程可以分为以下 3 步：

- (1) 获取用户输入的数据。
- (2) 验证用户的数据。
- (3) 返回提示信息。

例如，表单输入信息的验证、用户身份的验证、安全信息的验证、系统管理验证、技术操作验证等。其中与用户息息相关的主要是输入信息验证和用户身份验证，这些验证都是针对表单技术展开的。而有关更深层次的安全验证服务限于篇幅，本章就不涉及。

实现验证服务的途经可以包括客户端验证和服务器端验证。两者区别如下。

- ☑ 客户端验证：验证服务的脚本代码被发送到客户端浏览器中，当在浏览器中输入信息或者执行相关操作时，会触发这些本地脚本代码，然后对用户的操作以及输入的信息进行验证。客户端验证适合检测用户的操作行为是否恰当，用户输入的信息是否合法，而不能保证用户操作的准确性，以及输入信息的正确性。因为客户端验证的代码是公开的，任何人都可以在浏览器中查看到验证服务的脚本源代码。
- ☑ 服务器端验证：验证服务的脚本没有被发送到客户端，而是驻留在服务器端。当客户端 HTTP 请求把用户输入的信息发送到服务器端后，服务器端脚本将验证用户输入的信息是否正确。由于是在服务器端被执行，客户端看不见服务器端验证的脚本源代码，这对于保护 Web 应用程序的安全性具有重要作用，同时也能够保护用户的隐私。因此，服务器端验证适合检测用户身份是否正确，用户输入的信息是否安全、可靠。同时可以充分利用数据库来扩展验证服务的范围和能力。

通过 jQuery 向表单添加验证功能之前，必须记住一条重要原则：客户端验证不能够取代服务器端验证。同样，也不能够依赖用户启用 JavaScript。如果真想要求必须填写或者以特殊的格式填写某些字段，只靠 JavaScript 是不能够得到想要的结果的。有的用户喜欢禁用 JavaScript，有的设备可能不支持 JavaScript，而且少数用户还会绕过 JavaScript 的限制故意提交恶意数据。

实现验证服务的技术也有很多种，常用技术包括 JavaScript 等前台脚本、服务器端脚本以及 Ajax 技术。由于 JavaScript 与客户端验证对应，服务器端脚本与服务器端验证相对应，因此它们的功能和作用就不再重复介绍。Ajax 技术是最近几年流行的新技术，它能够结合客户端和服务器端，发挥前后台技术优势，把脚本放在客户端，而把数据放在服务器端，通过 XMLHttpRequest 组件实现在不刷新页面的情况下及时从后台读取数据，并借助这些数据在前台对信息进行验证。针对服务器环境下的常见验证服务流程图如图 11.8 所示。

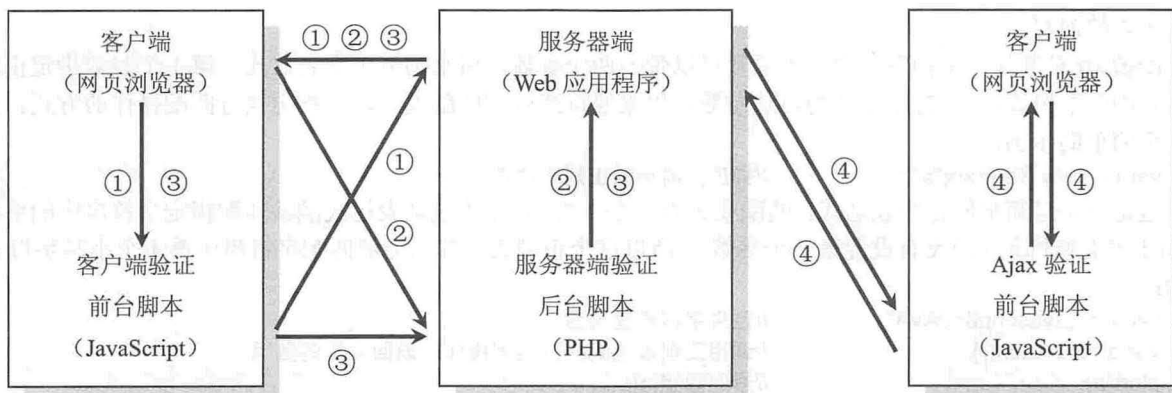


图 11.8 验证服务流程图

在图 11.8 的 Web 应用程序验证服务流程图中，存在 4 种常见的验证服务流程。具体说明如下：

- ☑ 第 1 种验证服务（如①线路），仅提供在客户端的验证，如果信息合法即被 HTTP 请求发送到服务器端的 Web 应用程序，经过服务器处理之后，把响应信息发送给客户端浏览器。这种服务主要验证提交信息是否合法，避免用户提交大量无用信息，给服务器处理造成不必要的负担，这样也避免了用户因为误输入信息而陷于盲目的等待之中。
- ☑ 第 2 种验证服务（如②线路），仅提供在服务器端的验证，用户信息提交之后，被 HTTP 请求发送到服务器端的 Web 应用程序，PHP 服务器首先调用验证脚本对提交的信息进行验证。如果通过验证则可以进一步进行处理或者执行下一步的操作，否则反馈错误信息。这种服务主要验证提交信息是否正确，常用来验证用户身份，或者验证输入的安全信息是否准确。
- ☑ 第 3 种验证服务（如③线路），它结合第 1 种和第 2 种验证服务，充分发挥这两种验证服务的优势，也是目前广泛使用的一种验证服务。
- ☑ 第 4 种验证服务（如④线路），是利用 Ajax 技术来实现的，也是目前比较时尚的验证技术。当用户输入信息并提交之后，首先在客户端进行处理，然后通过 XMLHttpRequest 组件把信息提交给服务器（在这个过程中，不需要发出 HTTP 请求），服务器端经过处理之后，再把验证的结果立即返回给浏览器（不经过刷新页面）。这样就能够做到，当注册新用户时，仅输入一个新用户名，就立即知道服务器端数据库中是否存在相同的用户名，而不需要在提交表单时，再等待服务器的答复，这样就不会浪费用户的劳动和时间。

11.2.2 认识正则表达式

正则表达式（Regular Expression）本质上是一种数学公式，用来描述字符模式的对象。正则表达式的应用非常广泛，特别是在字符串验证和处理方面。它的主要作用包括以下几点：

- ☑ 验证字符串，即验证给定的字符串或子字符串是否符合指定特征。例如，验证邮件地址、电话号码等用户提交数据是否合法等。
- ☑ 查找字符串，从指定的文本中查找符合指定特征的字符串，这比查找固定字符串更加灵活方便。
- ☑ 替换字符串，即把给定的字符串中的符合指定特征的字字符串替换为其他字符串，这比普通的替换更强大。
- ☑ 提取字符串，即从给定的字符串中提取符合指定特征的字字符串。

正则表达式使用比较简单，与 HTML、CSS、SQL 等标识语言一样，正则表达式也是规则不多的约定集合，但是熟练掌握这些规则，能够开发出功能强大的字符串处理程序。

JavaScript 通过内置的 Regular 对象实现对正则表达式的支持。RegExp 是 Regular Expression（正则表达

式) 短语的简写。

RegExp 对象是一个构造函数, 该函数可以带一两个参数, 用来构造正则表达式。第 1 个参数指定正则表达式的匹配模式, 第 2 个参数为可选参数, 用来修饰或限制匹配模式, 即指定执行匹配操作的方式。先看一个简单的示例:

```
var r = new RegExp("a");           //构造最简单的正则表达式
```

这是一个最简单的正则表达式, 匹配模式为一个字符 a, 即该正则表达式能够匹配指定字符串中的字符 a。由于没有修饰词 (即没有设置第 2 个参数), 所以这个正则表达式只能匹配字符串中第 1 个小写字母 a。例如:

```
var s = "javascript!=JAVA";        //定义字符串直接量
var a = s.match(r);                //调用正则表达式执行匹配操作, 返回匹配的数组
alert(a);                          //返回数组[a]
alert(a.index);                    //返回值为 1
```

上面的示例很直观地显示该正则表达式仅能够匹配字符串 javascript!=JAVA 中下标为 1 的字符 a, 后面的字母 a 将无法被匹配到 (即检索到)。

如果希望该正则表达式匹配字符串中所有的字母 a, 且不区分大小写, 则可以在第 2 个参数中增加 g 和 i 修饰词。例如:

```
var r = new RegExp("a","gi");      //设置匹配模式为全局匹配, 且不区分大小写
```

然后进一步试验:

```
var s = "javascript!=JAVA";        //字符串直接量
var a = s.match(r);                //匹配查找
alert(a);                          //返回数组["a","a","A","A"]
```

再回头仔细研究一下 RegExp 构造函数的两个参数。

首先, 第 2 个参数指定了正则表达式的修饰词。JavaScript 主要支持 g、i 和 m 这 3 个修饰性字符。具体说明如下:

- ☑ 字符 g 是 global (全局) 一词的缩写, 即用来指定全局匹配, 通俗地说就是正则表达式将在指定字符串范围内执行所有匹配查找, 而不是仅找到第 1 个匹配后就立即停止检索。
- ☑ 字符 i 是 case-insensitive (大小写不敏感) 短语中 insensitive 一词的缩写, 即执行不区分大小写的匹配, 也就是说对于大小写字母视为等同。
- ☑ 字符 m 是 multiline (多行) 一词的缩写, 即设置匹配模式能够在多行字符串中执行操作。

这 3 个修饰词分别指定了匹配操作的范围、大小写和多行行为。这些关键词可以自由组合, 以字符串的形式传递给 Regular() 构造函数。

在 ECMAScript 标准化之前, JavaScript 是不支持 m 修饰词的。实际上, 正则表达式可以支持的修饰词是非常多的, 如 x (设置空格匹配问题)、e (设置逆向匹配问题)、s (设置点号匹配问题) 等, 但是 JavaScript 不支持这些修饰词。

RegExp() 构造函数的第 1 个参数是一个字符串或者一个正则表达式, 该参数构成了正则表达式的匹配主体, 即匹配模式。不过它很讲究, 详细说明如下。

如果该参数是一个字符串, 则可以是一个符合正则表达式规则的字符串。例如:

```
var r = new RegExp("\\b\\w","gi");  //构造正则表达式对象
var s = "javascript JAVA";         //字符串直接量
var a = s.match(r);                //匹配查找
alert(a);                          //返回数组["j","J"]
```

字符串 \\b\\w 是一个匹配模式, 其中反斜杠表示转义序列, 双反斜杠表示斜杠本身的意思, 而 \\b 又表示单词的边界, \\w 表示任意 ASCII 字符, 所以上面的正则表达式将匹配字符串 javascript JAVA 中每个单词的首字母。

因为无论是字符串直接量, 还是正则表达式都使用字符 \\ 表示转义序列, 所以当将正则表达式作为字符串直接量传递给 RegExp() 构造函数时, 必须使用 \\ 替换所有 \\ 字符。

当要动态创建一个正则表达式,而不能使用正则表达式直接量的语法来表示时,构造函数 `RegExp()` 非常有用。例如,如果检索的字符串是由用户输入的,那么就必须在运行时使用 `RegExp()` 构造函数来创建正则表达式。

如果第 1 个参数是一个正则表达式,则第 2 个参数可以省略。这时 `RegExp()` 构造函数将使用与正则表达式参数相同的匹配模式和修饰词创建一个新的 `RegExp` 对象。例如:

```
var r = new RegExp("\\b\\w","gi");           //构造正则表达式对象
var r1 = new RegExp(r);                     //把正则表达式变量作为参数传递给 RegExp()构造函数
var s = "javascript JAVA";                 //字符串直接量
var a = s.match(r);                         //匹配查找
alert(a);                                  //返回数组["j","J"]
```

当然,很多时候可以把正则表达式直接量传递给 `RegExp()` 构造函数,以实现正则表达式进行类型封装。`RegExp()` 也可以作为普通的函数进行调用。如果不使用 `new` 运算符作为普通函数, `RegExp()` 函数的行为与使用 `new` 运算符调用构造函数时一样。不过如果函数的参数是正则表达式,那么它仅返回正则表达式,而不再创建一个新的 `RegExp` 对象。代码如下:

```
var a = new RegExp("\\b\\w","gi");           //构造正则表达式对象
var b = new RegExp(a);                     //对正则表达式对象进行再封装
var c = RegExp(a);                         //返回正则表达式直接量
alert(a.constructor == RegExp);           //返回 true
alert(b.constructor == RegExp);           //返回 true
alert(c.constructor == RegExp);           //返回 true
```

使用 `RegExp()` 构造函数创建正则表达式比较笨拙,因此可以使用直接量语法来创建 `RegExp` 对象。与创建函数直接量、对象直接量、数组直接量一样,定义正则表达式直接量非常简单和灵活。如果说引号是字符串直接量的语法分隔符,那么斜杠就是正则表达式直接量的语法分隔符。例如:

```
var r = /\b\w/gi;                           //定义正则表达式直接量
```

JavaScript 约定在正则表达式直接量中,双斜杠包含的字符为正则表达式的匹配模式,它是一组特殊的字符串,但是不能够使用引号进行包含。正则表达式的修饰词则跟随在最后一个斜杠的后面。例如,针对上面的正则表达式直接量,可以直接进行调用:

```
var s = "javascript JAVA";
var a = s.match(r);                         //直接调用正则表达式直接量
alert(a);                                  //返回数组["j","J"]
```

在 `RegExp()` 构造函数与正则表达式直接量语法中,匹配模式的表示是不同的。对于 `RegExp()` 构造函数,它接收的是字符串,而不是正则表达式的匹配模式本身。所以,在上面的示例中, `RegExp()` 构造函数中第 1 个参数中的特殊字符必须使用双反斜杠来表示,以防止字符串中每个字符被 `RegExp()` 构造函数转义。同时对于第 2 个参数中的修饰词也应该使用引号来包含。而正则表达式直接量中,每个字符都按正则表达式的规则来定义,普通字符与特殊字符都会被正确解释。

因此,可以在 `RegExp()` 构造函数中传递变量,而在正则表达式中是不允许的。例如:

```
var r = new RegExp("a"+ s + "b","g");       //动态创建正则表达式
var r = /"a"+ s + "b"/g;                     //错误的用法
```

在上面的示例中,对于正则表达式直接量来说,“`"`”和“`+`”都被视为普通字符进行匹配,而不是作为字符与变量的语法标识符来进行连接操作。

另外,正则表达式直接量的语法是 JavaScript 约定的,与其他语言中定义的规则可能会存在差异。例如,在 VBScript 脚本语言中,通过使用引号来定义正则表达式直接量,如“`^[\t]*$`”可以匹配一个空白行。

11.2.3 字符匹配

不管是使用 `RegExp()` 构造函数,还是使用正则表达式直接量,创建 `RegExp` 对象都比较容易。当然,如

何使用正则表达式的基本语法来描述字符的模式，就变得非常棘手，很多初学者在这里会逐渐迷失方向和兴趣。

1. 普通字符和元字符

根据正则表达式语法规则，匹配模式是由一系列字符构成的。大多数字符仅能够描述自身，这些字符称为普通字符，如所有的字母、数字等。也就是说，普通字符只能匹配字符串中与自身相同的字符。例如，对于字符串 javascript 来说，如果要查找整个字符串，需要这样设计：

```
var r = /javascript/;
```

显然，这种匹配模式与普通的字符查找没有两样，同时也失去了字符匹配的智能性和灵活性。所以，正则表达式还规定了一系列的特殊字符，特殊字符不是按照字符直接量进行匹配的，但是它们都拥有特殊的语义。例如：

(\) [] { } \ \ \ ^ \$ % ? * , .

特殊字符是不能够匹配自身的。由于具有特殊的语义，所以要匹配自身，还需要进行转义。正是得益于这些特殊的字符，用户才可以设计出很多精巧、灵活的正则表达式，并利用这些正则表达式匹配各种复杂的文本信息。因此，这些字符也称为元字符，即构造各种复杂文本的基本字符。

2. 字符直接量

字符的表示方法有多种（被表示的字符称为直接量），除了可以直接使用字符来表示它们本身外，还可以使用 ASCII 编码或者 Unicode 编码来表示。

☑ 用 ASCII 码表示

如果使用 ASCII 编码表示，必须指定一个两位的十六进制值，并在前面增加“\x”前缀。例如：

```
var r = /\x61/; //以 ASCII 编码匹配字母 a
```

由于字母 a 的 ASCII 编码为 97，转换为十六进制数值后为 61。因此如果要匹配字符 a，就应该在前面添加“\x”前缀，以提示该组合为 ASCII 编码。例如：

```
var s = "javascript";
var a = s.match(r); //即匹配第 1 个字符 a
```

除了十六进制外，还可以直接使用八进制数值来进行匹配。例如：

```
var r = /\141/; //141 是字母 a 的 ASCII 编码的八进制值
var s = "javascript";
var a = s.match(r); //即匹配第 1 个字符 a
```

使用十六进制需要添加“\x”前缀，主要是为了避免语义混淆，但是八进制不需要添加前缀。而对于十进制的 ASCII 编码值是不能够直接使用的。ASCII 编码只能匹配有限的单字节拉丁字符，对于双字节的汉字等字符是无法表示的。

☑ 用 Unicode 编码表示

如果使用 Unicode 编码表示，必须指定一个 4 位的十六进制值，并在前面增加“\u”前缀。例如：

```
var r = /\u0061/; //以 Unicode 编码匹配字母 a
var s = "javascript";
var a = s.match(r); //即匹配第 1 个字符 a
```

当这些特殊的转义字符用在 RegExp() 构造函数中时，应注意使用双斜杠来表示。例如：

```
var r = new RegExp("\\u0061");
```

因为，RegExp() 构造函数会把“\u”解释为字符 u 本身，所以对于“\u0061”编码来说，在构造的正则表达式中就成为了 u0061，从而失去它原来的语义。

除了这些固定的字符和转义编码外，JavaScript 还支持一些特殊的字符，如表 11.1 所示。

表 11.1 JavaScript 中的特殊字符

特殊字符	说明
\o	表示空字符，即 null，等价于\u0000（Unicode 十六进制编码）

续表

特殊字符	说 明
\t	表示制表符, 等价于\u0009
\n	表示换行符, 等价于\u000A
\v	表示垂直制表符, 等价于\u000B
\f	表示换页符, 等价于\u000C
\r	表示回车符, 等价于\u000D
\a	表示 alert 字符
\e	表示 escape 字符
\b	表示回退字符
\cX	表示控制字符^X, 如\cJ 等价于换行符\n

对于这些特殊字符, 如果在 RegExp 构造函数中使用, 必须使用双反斜杠进行转义。对于元字符, 如果要使用字符本义, 需要添加反斜杠前缀进行转义。例如:

```
\(, \), \[, \], \{, \}, \\\, \\\, \^, \$, \?, \*, \.
```

表示的语义分别是这些元字符本身, 此时它们就不再拥有特殊的语义:

```
(, ), [, ], {, }, \, |, ^, $, ?, *, .
```

如果无法确定某个字符是否拥有特殊的语义, 建议在每个字符前面都添加反斜杠前缀, 以进行转义, 匹配字符本身的意思。

考虑到许多字符在添加反斜杠前缀时, 也会拥有特殊的语义, 所以对于要直接匹配的字母和数字, 不应该使用反斜杠前缀进行转义, 如表 11.1 所示。当然在正则表达式中添加可以理解的反斜杠字符时, 必须使用反斜杠将其进行转义, 如正则表达式\\就表示匹配字符串中的反斜杠字符。

3. 定义简单字符类

所谓字符类, 就是把多个单独的字符放在中括号内构成一个字符集合。字符类能够与任何包含的字符进行匹配。例如:

```
var r = /[abc]/gi;           //定义字符类
var s = "javascript";        //字符串直接量
var a = s.match(r);          //返回数组["a", "a", "c"]
```

正则表达式/[abc]/gi 是一个简单的字符类, 它包含 3 个字符直接量: a、b、c。这样在字符串中只要包含字符类中任意一个字母, 都被视为匹配的意思。也就是说, 字符类相当于 JavaScript 的逻辑或运算, 中括号相当于逻辑或运算符 (||)。例如:

```
[abc]
```

可以理解为:

```
a || b || c
```

也就是说, 该字符类可以匹配字符 a、b 或 c, 但是不能够匹配字符 ab、bc、ab 或 abc。更通俗地讲, 字符类就相当于一个单选按钮组, 只能够匹配字符类中某个字符选项。例如, 对于下面这个字符串, 如果匹配所有单词 (共计 5 个), 使用简单的字符类就能够很轻松地解决:

```
var s = "abc abd abe abf abg";
```

简单分析该字符串, 每个词都包含有相同的子字符串 ab, 这样可以把每个词的第 3 个字符存放在中括号内, 构成一个字符类, 这样对于任何一个单词就都可以匹配了:

```
var r = /ab[cddefg]/g;        //通过简单的字符类实现匹配多个单词
var a = s.match(r);           //返回数组["abc", "abd", "abe", "abf", "abg"]
```

在字符类中 (即中括号内) 可以使用特殊字符。例如, 上面的正则表达式也可以这样设计:

```
var s = "abc abd abe abf abg";
var r = /ab[\u0063\u0064\u0065\u0066\u0067]/g; //通过字符的 Unicode 十六进制值来表示
var a = s.match(r);           //返回数组["abc", "abd", "abe", "abf", "abg"]
```


它们实现相同的匹配效果。

4. 定义反义字符类

所谓反义字符，就是除了字符类中指定的字符以外的任意字符。如果在字符类内部添加脱字符（^）前缀，即定义了反义字符类。例如，针对上面的正则表达式/ab[cdefg]/g，也可以这样设计：

```
var s = "abc abd abe abf abg";
var r = /ab[^ab]/g;           //通过反义字符类设计匹配模式
var a = s.match(r);           //返回数组["abc","abd","abe","abf","abg"]
```

在上面的匹配模式中，单词的第 2 个字符匹配除了 a 和 b 以外的任意字符。通过反义字符类，可以设计使用有限字符来匹配无限字符的意图。例如，在字符处理中，需要匹配的字符无法预料，或者难以通过简单字符类进行一一枚举，那么可以分析该匹配可能不会包含的字符，然后取反义，以反义字符类实现以少应多的目的。例如：

```
var r = /^[^0123456789]/g;
```

在这个正则表达式直接量中，将匹配除了数字以外的任意字符。反义字符类比简单字符类的功能更加强大和实用。

5. 定义字符范围类

如果匹配所有数字，可以这样设计：

```
var r = /^[0123456789]/g;
```

如果匹配所有字母，也可以这样设计：

```
var r = /[abcdefghijklmnopqrstuvwxyz]/gi;
```

此时可以使用字符范围类进行匹配。所谓范围类，就是指定字符列表时，仅指定起止字符，中间部分通过连字符（-）表示，从而达到简化表达式的目的。例如：

```
var r = /[0-9]/g;              //等价于 /^[0123456789]/g
var r = /[a-z]/g;              //等价于 /[abcdefghijklmnopqrstuvwxyz]/g
```

可以把它理解为数字 0~9 范围的任意数字匹配，或者是字母 a~z 之间任意字母匹配。注意，字符范围类是遵循字符编码的顺序来进行匹配的。如果所要测试的字符恰好是按照字符编码的排列顺序，就可以使用这种表达式来表示。

如果匹配任意 ASCII 字符，则可以这样设计：

```
var r = /[\u0000-\u00ff]/g;
```

如果匹配任意双字节的汉字，则可以这样设计：

```
var r = /[\u0000-\u00ff]/g;
```

即通过反义指定范围来实现匹配，因为双字节汉字是非常多的，借助反义范围能够快速达到目的。

对于数字来说，除了直接使用数字字符来表示外，还可以使用 Unicode 编码实现：

```
var r = /[\u0030-\u0039]/g;
```

当然，使用如下正则表达式可以匹配任意大写字母：

```
var r = /[\u0041-\u004A]/g;
```

或者使用如下正则表达式来匹配任意小写字母：

```
var r = /[\u0061-\u007A]/g;
```

也可以在字符类中混合使用简单字符、范围和反义等类型：

```
var s = "abcdez";              //字符串直接量
var r = /[abce-z]/g;           //字符类包含字符 a、b、c，以及从 e~z 之间的任意字符
var a = s.match(r);           //返回数组["a","b","c","e","z"]
```

注意，在字符类内部不要有空格，否则会被认为是一个空格进行匹配。例如：

```
var r = /[0-9 ]/g;
```

上面正则表达式不仅匹配所有数字，还会匹配所有空格。因此，如果要匹配任意大小写字母和数字，则可以使用：

第11章 表单开发

```
var r = /[a-zA-Z0-9]/g;
```

还可以在一个正则表达式中设计多个字符类，从而设计更灵活的匹配。例如：

```
var s = "abc4 abd6 abe3 abf1 abg7"; //字符串直接量
```

```
var r = /ab[c-g][1-7]/g;
```

//匹配第 1、2 个字符为 ab，第 3 个字符为 c~g，第 4 个字符为 1~7 的任意数字

```
var a = s.match(r); //返回数组["abc4","abd6","abe3","abf1","abg7"]
```

6. 预定义字符类

由于某些字符类比较常用，所以 JavaScript 的正则表达式语法就包含了一些特殊字符和转义序列来表示这些常用的类。例如，`\s` 匹配空格符、制表符和其他 Unicode 空白符，`\S` 匹配的是非 Unicode 空白符。预定义字符类的详细说明如表 11.2 所示。

表 11.2 预定义字符类

预定义字符类	说 明
.	匹配除了换行符和其他 Unicode 行终止符（如回车符）之外的任意字符，等价于 <code>[\^\\n\\r]</code>
<code>\w</code>	匹配任何 ASCII 单字符，等价于 <code>[a-zA-Z0-9_]</code>
<code>\W</code>	匹配任何非 ASCII 单字符，等价于 <code>[\^a-zA-Z0-9_]</code>
<code>\s</code>	匹配任何 Unicode 空白符，等价于 <code>[\t\\n\\x0B\\f\\r]</code>
<code>\S</code>	匹配任何非 Unicode 空白符，等价于 <code>[\^\\t\\n\\x0B\\f\\r]</code>
<code>\d</code>	匹配任何 ASCII 数字，等价于 <code>[0-9]</code>
<code>\D</code>	匹配任何非 ASCII 数字，等价于 <code>[\^0-9]</code>
<code>[b]</code>	匹配退格直接量（特例）
<code>[...]</code>	匹配位于括号内的任意字符
<code>[^...]</code>	匹配不在括号内的任意字符

有些字符类转义序列只匹配 ASCII 字符，还没有扩展到可以处理 Unicode 字符，读者可以自定义 Unicode 字符类。由于在方括号之内也可以使用这些特殊的字符类转义序列，因此可以使用`[\u0F00-\u0FFF]`匹配所有的藏文字符。使用预定义字符类可以简化正则表达式的表示模式。例如，匹配 3 个字母，如果不使用`\w`，则需要这样来设计：

```
var r = /[a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9]/g;
```

但是如果使用预定义字符类，正则表达式就会变得简单许多。例如，上面正则表达式可以这样来描述：

```
var r = /\w\w\w/g;
```

另外，转义序列`b` 具有特殊的语义，当它用在字符类中表示退格符，但是要在正则表达式中以直接量形式表示一个退格符，只需要使用具有一个元素的字符类（即`[b]`）来表示。

11.2.4 重复匹配

字符类能够精确匹配字符，但是却不能够描述字符匹配的次数。如果要匹配两个连续字符，可以描述为`\w\w/`；如果要匹配 3 个连续字符，可以进一步描述为`\w\w \w/`。但是如果匹配几十、上百个或者任意个字符，仅仅通过枚举字符类的形式来匹配就显得笨拙了。为此，正则表达式还定义了重复类匹配模式，它可以允许用户定义字符重复匹配的次数，其中包括确数或约数。

1. 简单重复性匹配

JavaScript 正则表达式约定了下面这些元字符，如表 11.3 所示。它们分别定义了字符重复匹配的确数或约数，也称为数量词或者量词。但它们的语义都是相同的，即设置匹配项可能重复显示的次数。

表 11.3 简单重复类

重 复 类	说 明
{n, m}	匹配前一项至少 n 次, 但是不能够超过 m 次
{n, }	匹配前一项至少 n 次, 或者更多次
{n}	匹配前一项恰好 n 次
?	匹配前一项 0 次或 1 次, 通俗说就是前一项是可选的, 等价于 {0, 1}
+	匹配前一项 1 次或多次, 等价于 {1, }
*	匹配前一项 0 次或多次, 等价于 {0, }

表 11.3 中所列的 6 种重复类用法比较灵活, 且语义有一定的重叠性, 使用时很容易混淆。下面结合一个具体的示例讲解。假设已创建了正则表达式来匹配下面一组字符串:

```
var s = "ggle gogle google gooogle goooogle goooooogle goooooooooogle  
goooooooooogle goooooooooogle"
```

☑ 如果仅匹配单词 ggle 和 gogle, 则可以这样设计:

```
var r = /go?gle/g; //匹配前一项字符 o 0 次或 1 次  
var a = s.match(r); //返回数组["ggle", "gogle"]
```

在这里元字符?表示前一项(单个字符或者子表达式)为可选项, 可有可无, 也就是说前面项没有也能够正确匹配。如果有, 则只能够匹配一次。对于这样的匹配结果, 还可以按如下正则表达式设计:

```
var r = /go{0,1}gle/g; //匹配前一项字符 o 0 次或 1 次  
var a = s.match(r); //返回数组["ggle", "gogle"]
```

大括号中第 1 个数值设置前一项最小重复次数, 0 表示它可以不出现, 1 表示它仅能够显示自身, 不能够重复显示。

☑ 如果仅匹配第 4 个单词 gooogle, 则可以这样设计:

```
var r = /go{3}gle/g; //匹配前一项字符 o 重复显示 3 次  
var a = s.match(r); //返回数组["gooogle"]
```

也可以通过简单的字符类来匹配:

```
var r = /gooogle/g; //匹配字符 gooogle  
var a = s.match(r); //返回数组["gooogle"]
```

当然这种方法显得很笨拙, 在重复项很多时, 用起来比较吃力。

☑ 如果希望匹配第 4~6 个之间的单词, 则可以这样设计:

```
var r = /go{3,5}gle/g; //匹配第 4~6 个之间的单词  
var a = s.match(r); //返回数组["ggle", "gogle"]
```

{3,5}分别指定了前一项字符 o 最少重复次数为 3, 最多重复次数为 5, 从而实现匹配第 4~6 个之间的 3 个单词。

☑ 如果希望匹配所有单词, 则可以这样设计:

```
var r = /go*gle/g; //匹配所有的单词  
var a = s.match(r); //返回数组["ggle", "gogle", "google", "gooogle", "goooogle",  
// "goooooogle", "gooooooogole", "goooooooooogle", "gooooooooooogole"]
```

其中元字符*表示前一项字符 o 可以不出现, 或者重复出现任意多次。当然, 还可以按如下方式进行设计:

```
var r = /go{0,}gle/g; //匹配所有的单词  
var a = s.match(r); //返回数组["ggle", "gogle", "google", "gooogle", "goooogle",  
"goooooogle", "gooooooogole", "goooooooooogle", "gooooooooooogole"]
```

{0,}指定前一项字符 o 可以出现 0 次, 或者任意多次。当{}中第 2 个参数值为空, 则表示任意值的意思。

☑ 如果希望匹配包含字符 o 的所有单词, 则可以这样设计:

```
var r = /go+gle/g;      //匹配的单词中字符 o 至少出现 1 次
var a = s.match(r);     //返回数组 ["gogle", "google", "gooogle", "gooogle", "goooooogle", "gooooooogle", "goooooooogle", "gooooooooogle"]
```

其中元字符+表示前一项字符 o 至少出现 1 次, 最多重复次数不限。当然, 还可以按如下方式进行设计:

```
var r = /go{1,}gle/g;   //匹配的单词中字符 o 至少出现 1 次
var a = s.match(r);     //返回数组 ["gogle", "google", "gooogle", "gooogle", "goooooogle", "gooooooogle", "goooooooogle", "gooooooooogle"]
```

重复类元字符总是出现在它们所作用的模式之后。使用重复类元字符*和?时要注意, 由于这些字符可能匹配前面字符或子表达式 0 次, 所以它们允许什么都不匹配。例如, 正则表达式/a*/实际上与字符串 bcd 匹配, 因为该字符串含有 0 个字符 a。

2. 贪婪匹配

正则表达式也具有贪婪性, 先从一个简单的示例说起:

```
var s = "<html><head><title></title></head><body></body></html>";
```

在上面这个字符串中, 如果希望匹配每个标签, 最简单的想法是按如下方法设计正则表达式:

```
var r = /<.*>/;
```

检测一下匹配结果:

```
var a = s.match(r);
alert(a);           //返回单元素数组["<html><head><title></title></head> <body>
</body></html>"]
```

看来正则表达式并没有按我们所想, 逐个匹配标签, 而是非常贪婪地把所有标签都匹配过来。这是因为星号(*)元字符在执行匹配时, 先看整个字符串是否匹配。如果不匹配, 则去掉该字符串中最后一个字符, 并再次尝试。如果还是没有发现匹配, 则再次去掉最后一个字符。如此递归计算, 直到发现一个匹配或者字符串不剩任何字符。所以, 上面示例中的正则表达式会先看整个字符串, 检测是否符合匹配标准。如果符合标准则执行匹配, 就返回整个字符串。到目前为止, 所有讨论的重复类元字符都具有贪婪的特性, 但是它们的贪婪表现不同。

☑ ?、{n} 和 {n, m} 重复类

?、{n} 和 {n, m} 重复类都具有弱贪婪性, 这种贪婪性主要表现为贪婪的有限性。例如:

```
var s = "<html><head><title></title></head><body></body></html>";
var r = /<.?/;
```

```
var a = s.match(r);      //返回单元素数组["<h"]
```

也就是说, 对于点号(.)元字符, 在选择匹配还是不匹配时, 如果条件允许, 它总会选择匹配, 而不是不匹配。对于{n}重复类, 它总是在允许的条件下, 匹配指定次数, 而不是不匹配或者少匹配。至于{n, m}重复类, 它总会在允许的条件下, 匹配 m 次, 而不是 n 次。例如, 下面的示例中, 正则表达式将匹配字符串"<html><head>", 而不是"<html>":

```
var r = /<.{4,10}>/
var a = s.match(r);      //返回单元素数组["<html><head>"]
```

但是, 如果按照下面模式进行设计, 正则表达式将匹配字符串<html>, 而不是<html><head>:

```
var r = /<.{4,9}>/
var a = s.match(r);      //返回单元素数组["<html>"]
```

这说明正则表达式的贪婪性是在遵循匹配条件基础上尽可能占有更多字符, 而不是随意占用。由于中间字符最大取 9 个字符, 则不符合匹配模式, 于是 JavaScript 解释器会丢掉一个字符, 再进行判断。如果不匹配, 继续丢掉最后一个字符, 直到符合匹配条件为止。

☑ *、+ 和 {n, } 重复类

*、+ 和 {n, } 重复类都具有强贪婪性, 这种贪婪性主要表现为贪婪的无限性。当然, 这种无限性也是在遵循匹配条件基础上实现尽可能的占有。不过, 这 3 个类表现也略有不同:

➤ 星号(*)重复类的匹配底线是最宽容的, 匹配欲望最强烈。不管是否存在指定字符或子表达

式都会执行匹配操作。

- 加号(+)重复类的匹配底线是最少存在一个符合指定条件的字符或子表达式, 否则不予执行匹配操作。
- {n, }重复类的匹配底线是最灵活的, 这种灵活性决定了它可以代替“*”或“+”重复类, 执行任意底线和条件的无限贪婪的匹配操作。例如, 使用{0, }代替“*”重复类, 使用{1, }代替“+”重复类, 或者自定义底线执行各种复杂的匹配操作。

正则表达式是有贪婪性的, 它总是与最长的可能长度匹配, 而且越是排在左侧的重复类匹配符优先级越高。例如:

```
var s = "<html><head><title></title></head><body></body></html>";
var r = /(<.*>)(<.*>)/
var a = s.match(r);
alert(a[1]);           //左侧匹配"<html><head><title></title></head><body></body>"
alert(a[2]);           //右侧子表达式匹配"</html>"
```

上面的演示示例说明, 当多个重复类并列在一起时, 左侧重复类具有较大优先权, 并尽可能占有更多的符合条件的字符, 但是它会留出最小匹配机会给右侧的重复类。再看一个示例:

```
var r = /(<.*>)(<.*>)(<.*>)/
var a = s.match(r);
alert(a[1]);           //左侧匹配<html><head><title></title></head><body>
alert(a[2]);           //右侧子表达式匹配</body>
alert(a[3]);           //右侧子表达式匹配</html>
```

上面的示例显示, 当多个重复类同时满足条件时, 会在保证右侧重复类最低匹配次数的基础上, 最左侧的重复类将尽可能占有所有字符。

3. 惰性匹配

与贪婪匹配相反, 惰性匹配遵循另一种算法。它先查看字符串中的第 1 个字符是否匹配。如果匹配条件不够, 就读入下一个字符。如果还是不能够匹配, 惰性匹配会继续从字符串中读取字符, 直到发现匹配或者整个字符串都检查过也没有匹配为止。

惰性匹配只需要在重复类后面添加问号(?)即可。这是 Perl 5 语言的一个特性, 在 JavaScript 1.5 和其后续版本中获得了支持。例如:

```
var s = "<html><head><title></title></head><body></body></html>";
var r = /<.*?>/
var a = s.match(r); //返回单元素数组["<html>"]
```

问号必须放在重复类字符后面。在上面示例中, JavaScript 解释器从字符串左侧开始逐个匹配, 经过 4 次迭代, 最终找到了匹配条件的子字符串<html>, 于是就停止迭代匹配。

如果说贪婪匹配体现了最大化匹配原则, 那么惰性匹配则体现最小化匹配原则。从语义角度分析, 它们属于反义词, 操作相反的两种行为。

但是, 读者也应该注意到, 不管是贪婪匹配, 还是惰性匹配, 虽然它们的算法不同, 但是它们都遵循这样的一条基本原则: 必须保证匹配满足正则表达式基本条件。例如, 在上面的示例中, 星号(*)可以不重复匹配任何字符, 也就是说, 对于正则表达式/<.*?>/来说, 它可以返回匹配字符串<>, 但是为了能够确保匹配条件成立, 在执行中还是匹配了带有 4 个字符的字符串 html。惰性取值不能够以违反基本匹配条件而返回, 除非没有找到符合条件的字符串, 否则必须满足它。

针对 6 种重复类的惰性匹配详细说明如下。

- ☑ {n, m}?: 正则表达式尽量匹配 n 次, 但是为了满足匹配条件也可能最多重复 m 次。
- ☑ {n}?: 正则表达式尽量匹配 n 次。
- ☑ {n, }?: 正则表达式尽量匹配 n 次, 但是为了满足匹配条件也可能匹配任意次。
- ☑ ??: 正则表达式尽量匹配, 但是为了满足匹配条件也可能最多匹配 1 次, 相当于{0, 1}?。

☑ `+`: 正则表达式尽量匹配 1 次, 但是为了满足匹配条件也可能匹配任意次, 相当于 `{1,}`。

☑ `*`: 正则表达式尽量不匹配, 但是为了满足匹配条件也可能匹配任意次, 相当于 `{0,}`。

当然, 使用惰性匹配时, 返回的结果会与用户期望的一致。例如:

```
var s = "<html><head><title></title></head><body></body></html>";
var r = /<.*?>/
var a = s.match(r);           //返回单元素数组["<html><head><title></title></head> <body>
</body></html>"]
```

对于正则表达式 `<.*>` 来说, 考虑到该模式将匹配字符 “<” 以及 0 个或多个字符, 然后跟随字符 “>”。在应用到字符串时, 它将匹配整个字符串。现在使用惰性匹配模式 `<.*?>`, 应该说它仅能够匹配字符串 `<html>`。但是在应用到上面的字符串时, 该模式也匹配整个字符串, 与贪婪匹配的结果是一样的。为什么这么说呢? 是因为正则表达式的模式匹配是在字符串中寻找第 1 个可能匹配的位置。惰性匹配在字符串的第 1 个字符处不匹配, 所以该匹配将返回, 甚至不考虑对后面的字符进行匹配。

4. 支配匹配

支配匹配是另一种类型的匹配模式, 它的算法是只尝试匹配整个字符串。如果整个字符串不能够匹配, 则会自动放弃匹配, 不再执行迭代以求进一步尝试。支配匹配只需要在重复类后面添加加号 (+) 即可。例如:

```
var s = "<html><head><title></title></head><body></body></html>";
var r = /<.*+>/
var a = s.match(r);           //返回单元素数组["<html><head><title></title></head> <body>
</body></html>"]
```

注意, 目前浏览器对支配匹配的支持不是很完善, IE 和 Opera 不支持该重复类量词, 而 FF 也仅是把它看做贪婪匹配。考虑到兼容的安全性, 一般不建议使用支配匹配模式。

11.2.5 高级匹配

正则表达式的语法定义是非常复杂的, JavaScript 语言也仅支持正则表达式的基本语法功能。下面介绍 JavaScript 所支持的高级匹配模式, 主要包括分组、选择、引用、前瞻和其他一些强大的正则表达式功能。掌握这些语法概念和用法, 就可以更方便地使用正则表达式进行复杂的字符串处理。

1. 分组

所谓分组, 就是通过使用小括号语法分隔符来包含一系列字符、字符类或重复类量词, 以实现处理各种特殊的字符序列。例如, 下面的字符串可以分拆出每个标签:

```
var s = "<html><head><title></title></head><body></body></html>";
```

如果使用贪婪模式进行匹配, 虽然可以抓取所有标签, 但是并没有劈开每个标签:

```
var r = /<.*>/
var a = s.match(r);           //返回单元素数组["<html><head><title></title></head> <body>
</body></html>"]
```

如果使用惰性模式进行匹配, 每次仅能够抓取一个标签:

```
var r = /<.*?>/
var a = s.match(r);           //返回单元素数组["<html>"]
```

但是, 如果利用分组来进行匹配, 就可以获取每个标签的名称:

```
var r = /(<.*?>)/g;           //分组模式
var a = s.match(r);           //全局匹配标签, 并存储到数组 a 中
for(var i = 0; i < a.length; i++) { //遍历数组 a, 获取每个标签的名称
    alert(a[i]);
}
```

在上面的示例中, 通过小括号逻辑分隔符, 实现分别存储每个被匹配的标签, 最后通过这个数组来获

取每个标签的名称。注意，对于正则表达式，小括号表示一个独立的逻辑域，其匹配的内容被独立存储，这样就可以以数组形式读取每个子表达式所匹配的信息。再看一个示例，假设准备匹配下面这个长字符串：

```
var s = "abcdef-abcdef-abcdef-abcdef-abcdef";
```

如果不使用分组，可能实现的正则表达式如下：

```
var r = /abcdef-abcdef-abcdef-abcdef-abcdef/;
```

```
var a = s.match(r); //返回单元素数组["abcdef-abcdef-abcdef-abcdef-abcdef"]
```

尽管这是可以的，但有点浪费。如果不知道该字符串中到底出现几次重复，可以使用分组来重写这个表达式：

```
var r = /(abcdef-?)*;/ //分组模式进行匹配
```

```
var a = s.match(r); //返回数组["abcdef-abcdef-abcdef-abcdef-abcdef", "abcdef"]
```

在小括号内的匹配模式表示正则表达式的子表达式，而跟随在小括号后的重复类数量词会作用于子表达式，而不是字符“)”。因此，上面的示例中通过小括号把每个标签内容作为匹配对象，然后通过重复类星号进行迭代匹配，最终能够快速实现匹配的目的。当然，并不限制在分组后使用星号，还可以使用任意重复类数量词：

```
var r = /(abcdef-?){5}/; //连续匹配 5 次子表达式
```

```
var r = /(abcdef-?){1,5}/; //最多匹配 5 次子表达式
```

```
var r = /(abcdef-?){0,}/; //匹配任意次子表达式
```

```
var r = /(abcdef-?)?/; //最多匹配一次子表达式
```

```
var r = /(abcdef-?)+/; //最小匹配一次子表达式
```

如果混合使用字符、字符类和量词，甚至可以实现一些相当复杂的分组。例如：

```
var s = "<html>< html><html >< html ></html>< /html>< / html>< / html >";
```

```
var r = /<([\\s]*)?html(\\s)*?>/g;
```

```
var a = s.match(r); //返回数组["<html >","< html >","< html >","< html >","</html >","< /html >","< / html >","< / html >"]
```

在上面的正则表达式中，使用了两个分组。第 1 个分组中包含了一个字符范围类，其中可以任意匹配空格或斜杠，文本范围类附加了一个量词*，它表示空格或斜杠可以出现任意次数。为了避免正则表达式的贪婪性，在重复类量词后面附加了问号(?)，令其以惰性模式进行匹配。第 2 个分组是一个简单的任意空格匹配，当然可以不要分组，但是对于第 1 个分组来说是必需的，因为数量词作用于子表达式，而不是某个特定的字符。通过上面正则表达式，可以匹配任意形式的<html>标签，这样就不用担心空格或斜杠对匹配语义的影响。

在正则表达式中，分组具有极高的应用价值。具体说明如下。

- ☑ 把单独的项目进行分组，以便合成子表达式，这样就可以像处理一个独立的字符那样，使用|、+、*或?等元字符来处理它们。例如：

```
var s = "javascript is not java";
```

```
var r = /java(script)?/g;
```

```
var a = s.match(r); //返回数组["javascript","java"]
```

上面的正则表达式可以匹配字符串 javascript，也可以匹配字符串 java，因为在匹配模式中通过分组，使用量词“?”来修饰该子表达式，这样匹配字符串时，其后既可以有 script，也可以没有。

- ☑ 在正则表达式中，通过分组可以在一个完整的模式中定义子模式。当一个正则表达式成功地与目标字符串相匹配时，也可以从目标字符串中抽出与小括号中的子模式相匹配的部分。例如：

```
var s = "ab=21,bc=45,cd=43";
```

```
var r = /(\\w+)=\\(d*)/;
```

```
var a = s.match(r); //返回数组["ab=21","ab","21"]
```

在上面的示例中，不仅要匹配出每个变量声明，而且希望知道每个变量的名称及其值。这时如果使用小括号进行分组，把需要独立获取的信息作为子表达式，就可以抽出声明，还可以提取更多有用的信息。

- ☑ 在同一个正则表达式的后部可以引用前面的子表达式。这是通过在字符“\”后加一位或多位数字实现的。数字指定了带括号的子表达式在正则表达式中的位置。如“\1”引用的是第 1 个带括号的

子表达式,“\2”引用的是第2个带小括号的子表达式。例如:

```
var s = "<h1>title<h1><p>text<p>";
var r = /(<V?\w+>).*\1/g;
var a = s.match(r); //返回数组["<h1>title<h1>", "<p>text<p>"]
```

在上面的示例中,通过引用前面子表达式匹配的文本,以实现成组匹配字符串。

由于子表达式可以嵌套在别的子表达式中,所以它的位置编号是根据左括号的顺序来定的。例如,在下面的正则表达式中,嵌套的子表达式(<V?\w+>)被指定为“\2”:

```
var s = "<h1>title<h1><p>text<p>";
var r = /(((<V?\w+>).*\2)/g;
var a = s.match(r); //返回数组["<h1>title<h1>", "<p>text<p>"]
```

注意,对正则表达式中前面子表达式的引用,所指的并不是那个子表达式的模式,而是与那个模式相匹配的文本。例如,下面这个字符串就无法实现匹配:

```
var s = "<h1>title</h1><p>text</p>";
var r = /(((<V?\w+>).*\2)/g;
var a = s.match(r); //返回 null
```

虽然子表达式(<V?\w+>)可以匹配<h1>,也可以匹配</h1>,但是对于\2来说,它引用的是前面子表达式匹配的文本,而不是它的匹配模式。如果要引用前面子表达式的匹配模式,则必须使用下面的正则表达式:

```
var r = /(((<V?\w+>).*(<V?\w+>))/g;
var a = s.match(r); //返回数组["<h1>title</h1>", "<p>text</p>"]
```

2. 引用

实际上在正则表达式执行匹配运算时,表达式计算会自动把每个分组(子表达式)匹配的文本都存储在一个特殊的地方以备将来使用。这些存储在分组中的特殊值,称为反向引用。反向引用将遵循从左到右的顺序,根据表达式中的左括号字符的顺序进行创建和编号。例如:

```
var s = "abcdefghijklmn";
var r = /(a(b(c)))/;
var a = s.match(r); //返回数组["abc", "abc", "bc", "c"]
```

在这个分组匹配模式中,共产生了3个反向引用,第1个是“(a(b(c)))”,第2个是“(b(c))”,第3个是“(c)”。它们引用的匹配文本分别是字符串 abc、bc 和 c。

反向引用在应用开发中主要包含以下几种常规用法。

- ☒ 在正则表达式对象的 test()方法以及字符串对象的 match()和 search()等方法中使用。在这些方法中,反向引用的值可以从 RegExp()构造函数中获得。例如:

```
var s = "abcdefghijklmn";
var r = /(\w)(\w)(\w)/;
r.test(s);
alert(RegExp.$1); //返回第1个子表达式匹配的字符 a
alert(RegExp.$2); //返回第2个子表达式匹配的字符 b
alert(RegExp.$3); //返回第3个子表达式匹配的字符 c
```

通过上面的示例可以看到,正则表达式执行匹配测试后,所有子表达式匹配的文本都被分组存储在 RegExp()构造函数的属性内,通过前缀符号\$与正则表达式中子表达式的编号来引用这些临时属性。其中属性\$1标识符指向第1个值引用,属性\$2标识符指向第2个值引用,依此类推。

- ☒ 可以直接在定义分组的表达式中包含反向引用。这可以通过使用特殊转义序列(如“\1”、“\2”等)来实现。例如:

```
var s = "abcbcacba";
var r = /(\w)(\w)(\w)\2\3\1\3\2\1/;
var b = r.test(s); //验证正则表达式是否匹配该字符串
alert(b); //返回 true
```

在上面示例的正则表达式中,“\1”表示对第1个反向引用(\w)所匹配的字符 a 引用,“\2”表示对第

2 个反向引用(\w)所匹配的字符 b 引用,“\3”表示对第 3 个反向引用 (\w) 所匹配的字符 c 引用。

- ☑ 可以在字符串对象的 `replace()` 方法中使用。通过使用特殊字符序列 \$1、\$2、\$3 等来实现。例如, 在下面的示例中将颠倒相邻字母和数字的位置。代码如下:

```
var s = "aa11bb22c3d4e5f6";
var r = /(\w+?)(\d+)/g;
var b = s.replace(r,"$2$1");
alert(b);           //返回字符串 11aa22bb3c 4d5e6f
```

在这个例子中, 正则表达式包括两个分组, 第 1 个分组匹配任意连续的字母, 第 2 个分组匹配任意连续的数字。在 `replace()` 方法的第 2 个参数中, \$1 表示对正则表达式中第 1 个子表达式匹配文本的引用, 而 \$2 表示对正则表达式中第 2 个子表达式匹配文本的引用。通过颠倒 \$1 和 \$2 标识符的位置, 即可实现字符串的颠倒替换原字符串。

3. 非引用型分组

正则表达式分组会占用一定的系统资源, 在较长的正则表达式中, 存储反向引用会降低匹配速度。但是很多时候使用分组仅是为了设置操作单元, 而不是为了引用, 这时候建议选用一种非引用型分组, 它不会创建反向引用。

通过使用非引用型分组, 既可以拥有与匹配字符串序列同样的能力, 又不用存储匹配文本的开销。创建非引用型分组的方法是在左括号的后面分别加上一个问号和冒号。例如:

```
var s1 = "abc";
var s2 = "123";
var r = /(?:\w*?)(?:\d*?)/;    //非引用型分组
var a = r.test(s1);           //返回 true
var b = r.test(s2);           //返回 true
```

此时如果调用 `RegExp` 对象的 \$1 标识符来引用分组匹配的文本信息, 结果会返回一个空字符串, 因为该分组是非引用型的。如下:

```
alert(RegExp.$1);           //返回 ""
```

正因为如此, 字符串对象的 `replace()` 方法就不能通过 `RegExp.$1` 变量来使用任何反向引用, 或在正则表达式中使用它。非引用型分组对于必须使用子表达式, 但是又不希望存储无用的匹配信息而浪费系统资源, 或者希望提高匹配速度, 是非常重要的方法。

4. 选择

当正则表达式执行匹配操作时, 经常会遇到选择性匹配问题。例如, 如果希望匹配字符串 abc, 同时还要匹配字符串 123。由于这两个字符串完全没有相同的字符, 按照前面介绍的方法, 需要编写两个不同的正则表达式, 并分别对两字符串进行匹配。代码如下:

```
var s1 = "abc";
var s2 = "123";
var r1 = /\w+/;
var r2 = /\d+/;
var b1 = r1.test(s1);
var b2 = r2.test(s2);
alert(b1);           //返回 true
alert(b2);           //返回 true
```

当然, 也可以使用字符范围类, 实现相同的匹配选择操作。例如:

```
var s1 = "abc";
var s2 = "123";
var r = /[(\w+)(\d+)]/;      //字符范围类正则表达式
var b1 = r.test(s1);         //返回 true
var b2 = r.test(s2);         //返回 true
```

在正则表达式的字符范围类中,把两个匹配模式分别包含在小括号语法分隔符中,组成两个独立的子表达式,并通过范围类进行匹配操作。不过,这种方式的执行效率比较低,正则表达式会为不同的子表达式开辟内存空间,这样就占用了不必要的系统资源。

JavaScript 正则表达式提供对选择操作符的支持,选择操作符使用管道符(|)表示,把它放在两个独立的单元之间,表示对多个单元执行选择匹配操作。例如:

```
var r = /a|b/;           //选择单个字符, 匹配 a 或 b
var r = /\w*\d+/;        //选择重复类, 匹配任意字符, 或者匹配任意数字
var r = /a\d+/;          //选择字符或重复类, 匹配字符 a, 或者匹配任意数字
var r = /\w*\d{1,}?/;     //选择惰性匹配
var r = /a\d{1,}?/;       //选择字符或惰性模式
var r = /[abc][def]/;     //选择字符范围类
var r = /a[cde]/;         //选择字符或者字符范围类
var r = /(abc)|(cdf)/;    //选择子表达式
```

针对上面的示例,如果使用选择操作符,则可以这样设计:

```
var s1 = "abc";
var s2 = "123";
var r = /\w*\d+/;         //选择重复字符类
var b1 = r.test(s1);      //返回 true
var b2 = r.test(s2);      //返回 true
```

也可以设计多重选择模式,这时只需要在多个连续的单元之间加入选择操作符即可,执行连续选择匹配操作,例如:

```
var s1 = "abc";
var s2 = "efg";
var s3 = "123";
var s4 = "456";
var r = /(abc)|(efg)|(123)|(456)/; //多重选择匹配
var b1 = r.test(s1);               //返回 true
var b2 = r.test(s2);               //返回 true
var b3 = r.test(s3);               //返回 true
var b4 = r.test(s4);               //返回 true
```

在实际开发中经常会使用选择操作符。例如,针对提交的表单信息进行敏感字符过滤,这时可以设计一个匹配所有敏感字符的正则表达式,然后使用字符串对象的 `replace()` 方法把所有敏感字符替换为字符编码,并转换为网页显示的编码格式。代码如下:

```
var s = "a?b?c&";           //待过滤的表单提交信息
var r = /\|\"|\?|\&/gi;      //过滤敏感字符的正则表达式
function f(){                 //替换函数, 把敏感字符替换为对应网页显示的编码格式
    return "&#" + arguments[0].charCodeAt(0) + ";";
}
var a = s.replace(r,f);       //执行过滤替换
document.write(a);            //在网页中显示正常的字符信息
alert(a);                     //返回字符串 "a&#39;b&#63;c&#38;"
```

最后,把 JavaScript 正则表达式的分组、引用和选择元字符进行汇总说明,以方便读者比较参考,如表 11.4 所示。

表 11.4 JavaScript 正则表达式的分组、引用和选择元字符

元 字 符	说 明
	选择。匹配的是该符号左边的子表达式或右边的子表达式
(...)	组合。将多个项目组合为一个单元,这个单元可由*、+、?和 等元字符使用,而且还可以存储与这个组合所匹配的字符,以供此后的引用使用

元 字 符	说 明
(?:...)	非引用型组合。把项目组合到一个单元，但是不记忆与该组匹配的字符
\n	引用。引用第 n 个分组第 1 次匹配的字符。组是括号中的子表达式（可能是嵌套的）。编号是从左到右计数的左括号数，不过以(?:形式分组的组不被编码

5. 声明

在 JavaScript 正则表达式中，可以使用任意正则表达式作为锚定条件，用来声明正则表达式在什么条件下才能匹配，或者不在什么条件下才会匹配。这种声明包括正前向声明和反前向声明两种模式。

- ☑ 所谓正前向声明，这里的前向是指定匹配模式后面的字符，声明表示条件的意思，也就是指定接下来的字符必须被匹配，但并不真正进行匹配。通俗地说，就是指定可能执行匹配操作的条件，该条件作为正则表达式匹配模式的一部分而存在，但是不会真正使用该条件去执行匹配。

正前向声明使用(?:=匹配条件)来表示。例如：

```
var s = "a:123 b=345";
var r = /\w*(?=)/;           //使用正前向声明，指定执行匹配必须满足的条件
var a = s.match(r);          //返回数组["b"]
```

在上面的示例中，通过使用(?:=)锚定条件，指定只有在\w*所能够匹配的字符后面跟随一个等号字符才能够执行\w*匹配。所以，最后匹配的是字符 b，而不是字符 a。

- ☑ 所谓反前向声明，就是与正前向声明匹配相反，指定接下来的字符都不必匹配。反前向声明使用(?:!=匹配条件)来表示。例如：

```
var s = "a:123 b=345";
var r = /\w*(?!)/;           //使用反前向声明，指定执行匹配不必满足的条件
var a = s.match(r);          //返回数组["a"]
```

在上面的示例中，通过使用(?:!=)锚定条件，指定只有在\w*所能够匹配的字符后面不跟随一个等号字符，才能够执行\w*匹配。所以，最后匹配的是字符 a，而不是字符 b。

声明虽然包含在小括号内，但这不是分组。事实上，分组不会考虑声明的存在。也就是说，对于正则表达式，它们仅是一个修饰性限定条件，而不是可引用的单元。

此时如果调用 RegExp 对象的 \$1 标识符来引用分组匹配的文本信息，结果会返回一个空字符串，因为这里的小括号不是分组标志。代码如下：

```
alert(RegExp.$1);           //返回""
```

目前，JavaScript 仅支持前向声明，不支持后向声明。也就是说，能够根据前面的字符是否匹配某个指定的表达式，来决定执行后面的匹配操作。

6. 边界

JavaScript 正则表达式支持定位功能，所谓定位就是能够确定字符在字符串中的具体方位（如字符串的头部或尾部，或者单词的边界）。详细说明如表 11.5 所示。

表 11.5 边界元字符

元 字 符	说 明
^	匹配主犯策划的开头，在多行检测中，会匹配一行的开头
\$	匹配主犯策划的结尾，在多行检测中，会匹配一行的结尾
\b	匹配一个词语的边界。具体说就是位于字符\w 和\w 之间的位置，或者位于字符\w 和字符串的开头和结尾之间的位置，但是\b匹配的是退格符，而不是边界
\B	匹配非词语边界的字符

利用这 4 个边界元字符，可以很轻松地定位所要匹配的字符在文本行中的位置。

☑ 如果匹配文本行中最后一个单词，则可以使用如下方法实现：

```
var s = "how are you";
var r = /(?:\w+)$/;
var a = s.match(r);    //返回数组["you"]
```

☑ 如果匹配文本行中开头一个单词，则可以使用如下方法实现：

```
var s = "how are you";
var r = /^(?:\w+)/;
var a = s.match(r);    //返回数组["how"]
```

☑ 如果匹配文本行中每一个单词，则可以使用如下方法实现：

```
var s = "how are you";
var r = /(?:\w+)/g;
var a = s.match(r);    //返回数组["how", "are", "you"]
```

如果使用下面的方法，也会匹配每个单词，但是它也会把单词之间的空格作为一个单词进行匹配：

```
var r = /\b(?:.+?)\b/g;
var a = s.match(r);    //返回数组["how", "", "are", "", "you"]
```

不过也可以这样设计：

```
var r = /\b(?:S+?)\b/g;    //匹配非边界区域的字符
var a = s.match(r);    //返回数组["how", "are", "you"]
```

请注意，在匹配单个字符串时，可以在匹配模式中添加行边界。例如：

```
var s = "javascript";
var r = /^javascript$/;    //在匹配模式中添加行边界
var b = r.test(s);    //返回 true
```

当然也可以不添加边界，这样不会影响执行匹配的结果。

从语法角度上分析，这种单词间必须添加空格的格式是正确的，但是在正则表达式中就会引发匹配错误。例如，检索单词 javascript 自身，可以使用如下模式：

```
var r = /sjavascript\s/;
```

也就是说，在单词前后都要有空格，但是这样做存在两个潜在问题：

☑ 如果单词出现在字符串的开头或结尾，该匹配模式就会失效。除非在字符串的开头或结尾有一个空格。

☑ 当该模式匹配字符串时，在返回的匹配字符串前后都会带有空格。

因此，应该把空格和单词边界区分开来，正确选择单词的边界匹配符**\b**来代替空格匹配符**\s**进行匹配。

7. 锚记

JavaScript 正则表达式定义了很多元字符，在这些元字符中包含一类特殊的字符，它们不匹配具体的字符，而是匹配字符之间的位置。例如，**\b**元字符匹配的是单词的边界，也就是说位于**\w**元字符（匹配 ASCII 单字符）与**\w**元字符之间的边界，或者位于一个 ASCII 单字符与一个字符串的开头或结尾之间的边界。类似**\b**这样的元字符，也称为正则表达式的锚记。

正则表达式的锚记与 HTML 锚记概念有点类似，都具有定位功能。但正则表达式的锚记可以将匹配模式定位在检索字符串中的一个特定位置上。最常用的锚记是[^]，它能够使模式定位在字符串的开头，而锚记^{\$}能够使模式定位在字符串的末尾。JavaScript 支持的锚记如表 11.6 所示。

表 11.6 锚记

锚 记	说 明
[^]	匹配主犯策划的开头，在多行检测中，会匹配一行的开头
^{\$}	匹配主犯策划的结尾，在多行检测中，会匹配一行的结尾
^{\b}	匹配一个词语的边界。具体说就是位于字符 \w 和 \w 之间的位置，或者位于字符 \w 和字符串的开头和结尾之间的位置，但是 ^[b] 匹配的是退格符，而不是边界

续表

锚 记	说 明
\B	匹配非词语边界的字符
(?=P)	正前向声明, 要求接下来的字符都与模式 P 匹配, 但是不包括匹配中的那些字符
(?!P)	反前向声明, 要求接下来的字符不与模式 P 匹配

8. 标志

正则表达式的标志说明了高级模式匹配的规则。与其他的正则表达式语法不同, 标志是在模式分隔符/之外进行说明的, 它不会出现在模式分隔符(即两个斜杠)之间, 而是位于第 2 个斜杠之后。JavaScript 正则表达式支持的标志包括 3 种, 说明如表 11.7 所示。

表 11.7 JavaScript 正则表达式支持的标志

标 志	说 明
i	执行大小写不敏感的匹配
g	执行全局匹配, 即找到所有匹配, 而不是在找到第 1 个之后就停止
m	执行多行匹配, 其中使用^可以匹配一行的开头和字符串的开头, 使用\$可以匹配一行的结尾或字符串的结尾

表 11.7 中的 3 个标志可以任意混合使用。例如, 如果把标志 i 与 g 混合使用, 就可以执行一个全局的、大小写不敏感的匹配。例如:

```
var r = /javascript/gi;
```

上面的模式可以匹配指定字符串中所有的 javascript、JAVASCRIPT、Javascript、javaScript 或 JavaScript 等不同大小写的字符串。

在 JavaScript 1.5 版本以前, 由于不支持 m 标志, 为了实现多行匹配, 需要使用 split()方法将字符串分割成行与行的数组, 然后再对每一行单独进行正则表达式测试。看下面的示例, 它只能够匹配最后一个字母 c, 而前面两个字母没有被匹配, 因为只有字母 c 在字符串的结尾处:

```
var s = "a\nb\nc";
var r = /\w+$/g;           //默认为单行匹配
var a = s.match(r);        //返回数组["c"]
```

由于字符串 s 中有两个换行符, 如果匹配每一行结尾的字母, 则需要使用多行匹配模式:

```
var s = "a\nb\nc";
var r = /\w+$/gm;          //启动多行匹配
var a = s.match(r);        //返回数组["a", "b", "c"]
```

多行模式会改变边界的匹配行为, 这时\$会匹配换行符之前的位置, 而不再是字符串的结尾。

11.2.6 匹配操作

在 JavaScript 中, 每个正则表达式都是一个对象。正则表达式是使用 RegExp 对象来表示的, 除了 RegExp()构造函数外, RegExp 对象还拥有很多属性和方法, 熟练掌握这些方法和属性的使用, 能够提高正则表达式的应用技巧。

1. 实例属性

RegExp 类比较特殊, 它不仅定义了类(即静态)属性, 还定义了实例属性。也就是说, 它既定义了属于构造函数 RegExp()的全局属性, 又定义了属于 RegExp 对象的具体实例属性。正则表达式的实例属性如表 11.8 所示。

表 11.8 正则表达式的实例属性

实 例 属 性	说 明
global	返回 Boolean 值, 设置正则表达式是否全局匹配, 可以表示标志 g (全局选项) 是否已经设置
ignoreCase	返回 Boolean 值, 设置正则表达式是否区分大小写, 可以表示标志 i (忽视大小写选项) 是否已经设置
multiline	返回 Boolean 值, 设置正则表达式是否多行匹配, 可以表示标志 m (多行模式选项) 是否已经设置
lastIndex	返回整数值, 表示下次匹配的起始位置, 只有调用 exec() 或 test() 方法时有效
source	返回正则表达式的源字符串文本

对于任何一个正则表达式对象, 它都会拥有这 5 个属性。不过由于可以使用标志来设置正则表达式的这些属性, 所以一般很少使用它们。例如:

```
var r = /a/gi;           //声明正则表达式直接量
alert(r.global);         //返回 true
alert(r.ignoreCase);     //返回 true
alert(r.multiline);      //返回 false
alert(r.source);         //返回 a
```

注意, global、ignoreCase、multiline 和 source 属性都是只读属性。

lastIndex 属性比较有用, 它是可以读写的。对于具有标志 g 的匹配模式来说, 该属性存储了在字符串中下一次开始检索的位置。它仅与 exec() 和 test() 方法配合使用。例如:

```
var s = "javascript is not java";
var r = /a/gi;           //正则表达式直接量
r.exec(s);               //第 1 次执行匹配
alert(r.lastIndex);      //返回值为 2
r.exec(s);               //第 2 次执行匹配
alert(r.lastIndex);      //返回值为 4
r.exec(s);               //第 3 次执行匹配
alert(r.lastIndex);      //返回值为 20
r.exec(s);               //第 4 次执行匹配
alert(r.lastIndex);      //返回值为 22
r.exec(s);               //第 5 次执行匹配
alert(r.lastIndex);      //返回值为 0
```

在上面的示例中, 正则表达式 r 查找字母 a。当它首次检测时, 发现在第 2 个位置 (即序号为 1) 有一个字母 a, 于是 lastIndex 属性就被设置为 2, 即开始下一次匹配时的起始位置。当再次调用 exec() 方法时, 就会从 lastIndex 属性指定的位置开始匹配, 依此类推。可以手动改变 lastIndex 属性值, 强迫正则表达式从指定的位置开始执行检测。例如:

```
var s = "0123456789";
var r = /\d/g;           //匹配单个数字
r.lastIndex = 5;         //指定匹配起始位置为 5, 即从第 6 个字符开始匹配
var a = r.exec(s);       //执行匹配
alert(a);                //返回匹配数字为 5
```

2. 静态属性

正则表达式的静态属性比较特殊, 因为它们都有两个名字: 长名 (全称) 和短名 (简称, 以美元符号开头表示)。详细说明如表 11.9 所示。

表 11.9 静态属性

静 态 属 性	短 名	说 明
input	\$	最后用于匹配的字符串, 即传递给 exec() 或 test() 方法的字符串
lastMatch	\$&	最后匹配的字符

续表

静 态 属 性	短 名	说 明
lastParen	\$+	最后匹配的分组
leftContext	\$`	在上次匹配之前的子字符串
multiline	\$*	用于指定是否所有表达式都使用多行模式的布尔值
rightContext	\$'	在上次匹配之后的子字符串

在下面的这个示例中，匹配字符串 JavaScript，不区分大小写。代码如下：

```
var s = "JavaScript,not Javascript";
var r = /(Java)Script/gi;
var a = r.exec(s);           //执行匹配操作
alert(RegExp.input);         //返回字符串 JavaScript,not Javascript
alert(RegExp.leftContext);   //返回空字符串，因为第 1 次匹配操作时，左侧没有内容
alert(RegExp.rightContext);  //返回字符串,not Javascript
alert(RegExp.lastMatch);     //返回字符串 JavaScript
alert(RegExp.lastParen);     //返回字符串 Java
```

该示例演示了正则表达式的几个静态属性的用法。

- ☑ input 属性实际上存储的是被执行匹配的字符串，即整个字符串 JavaScript,not Javascript。
- ☑ leftContext 属性存储的是执行第 1 次匹配之前的子字符串，这里为空，因为在第 1 次匹配的文本 JavaScript 左侧为空。而 rightContext 属性存储的是执行第 1 次匹配之后的子字符串，即为 “,not Javascript”。
- ☑ lastMatch 属性包含的是第 1 次匹配的子字符串，即为 JavaScript。
- ☑ lastParen 属性包含的第 1 次匹配的分组，即为 Java。如果模式中包含多个分组，则会显示最后一个分组所匹配的字符。例如：

```
var r = /(Java)(Script)/gi;
var a = r.exec(s);           //执行匹配操作
alert(RegExp.lastParen);     //返回字符串 Script，而不再是 Java
```

当然，也可以使用短名来读取这些属性所包含的值，考虑到这些短名不符合 JavaScript 语法规则，则必须使用中括号运算符来进行读取操作。不过对于 \$_ 属性来说，它符合 JavaScript 标识符语法规则，因此可以直接使用。例如，针对上面示例也可以这样设计：

```
var s = "JavaScript,not Javascript";
var r = /(Java)(Script)/gi;
var a = r.exec(s);
alert(RegExp.$_);           //返回字符串 JavaScript,not Javascript
alert(RegExp["$`"]);         //返回空字符串
alert(RegExp["$'"]);         //返回字符串,not Javascript
alert(RegExp["$&"]);         //返回字符串 JavaScript
alert(RegExp["$+"]);         //返回字符串 Java
```

当然，这些属性的值都是动态的，每次执行 exec() 或 test() 方法时，所有属性值都会被重新设置。例如，在下面示例中第 1 次执行匹配和第 2 次执行匹配时，这些静态属性值都会实时动态更新：

```
var s = "JavaScript,not Javascript";
var r = /Scrip(t)/gi;        //第 1 次定义的匹配模式
var a = r.exec(s);          //执行第 1 次匹配
alert(RegExp.$_);           //返回字符串 JavaScript,not Javascript
alert(RegExp["$`"]);         //返回字符串 Java
alert(RegExp["$'"]);         //返回字符串,not Javascript
alert(RegExp["$&"]);         //返回字符串 Script
alert(RegExp["$+"]);         //返回字符串 t
```

```

var r = /Jav(a)/gi;           //第2次定义的匹配模式
var a = r.exec(s);           //执行第2次匹配
alert(RegExp.$_);            //返回字符串 JavaScript,not Javascript
alert(RegExp["$"]);          //返回空字符串
alert(RegExp["$"]);          //返回字符串 Script,not Javascript
alert(RegExp["$&"]);         //返回字符串 Java
alert(RegExp["$+"]);         //返回字符串 a

```

通过上面的示例可以看出, RegExp 对象的静态属性是公共的, 对于所有正则表达式来说都可以共享, 因此它们的值也是实时变化的。multiline 属性与上面几个属性不同, 它不会根据每次执行的操作进行实时更新, 而且它还可以控制所有正则表达式的 m 标志项。例如:

```

var s = "a\nb\nc";
var r = /w+$/g;               //定义匹配模式
var a = s.match(r);          //执行默认匹配, 返回数组["c"]
RegExp.multiline = true;     //动态设置模式为多行匹配
var a = s.match(r);          //返回数组["a", "b", "c"]

```

IE 和 Opera 浏览器不支持 RegExp.multiline 属性, 考虑到浏览器的兼容性, 不建议使用这种动态方式设置正则表达式的多行匹配模式。

3. 实例方法

RegExp 对象定义的 3 个用于执行模式匹配操作的方法, 如表 11.10 所示。它们的行为与 String 对象的正则表达式操作方法类似。例如, RegExp 对象的 exec() 方法与 String 对象的 match() 方法操作相似, 只不过 exec() 是以字符串为参数的 RegExp 对象方法, 而 match() 方法是以正则表达式为参数的 String 对象方法。在非全局模式下, 它们的返回值是相同的。

表 11.10 RegExp 对象定义的 3 个方法

实 例 方 法	说 明
exec()	通用的匹配模式
test()	检测一个字符串是否匹配某个模式
toString()	把正则表达式转换成字符串

(1) exec() 方法

作为正则表达式的通用匹配方法, exec() 方法比 RegExp.test()、String.search()、String.replace() 和 String.match() 都复杂。该方法需要一个参数, 用来执行要执行操作的字符串, 并返回一个数组, 数组中存放的是匹配结果。如果没有找到匹配, 返回值为 null。例如:

```

var s = "javascript";
var r = /java/g;
var a = r.exec(s);           //返回数组["java"]

```

exec() 方法的工作机制是这样的: 当调用方法时, 先检索字符串参数, 从中获取与正则表达式相匹配的文本。如果找到了匹配的文本, 就会返回一个数组; 否则, 返回 null。返回数组元素的具体说明如下。

- ☑ 第 0 个元素: 是与表达式相匹配的文本。
- ☑ 第 1 个元素: 是与正则表达式的第 1 个子表达式相匹配的文本 (如果存在)。
- ☑ 第 2 个元素: 是与正则表达式的第 2 个子表达式相匹配的文本, 依此类推。

返回数组还包含几个属性, 具体说明如下。

- ☑ length: 该属性声明的是数组中的元素个数。
- ☑ index: 该属性声明的是匹配文本的第 1 个字符的位置。
- ☑ input: 该属性包含的是整个字符串。

当调用非全局模式的正则表达式对象的 exec() 方法时, 返回的数组与调用字符串对象的 match() 方法返

回数组是完全相同的，即它们都将进行检索，并返回上述元素和属性。

当执行全局匹配模式时，`exec()`的行为就略有变化。这时它定义 `lastIndex` 属性，用来指定下一次执行匹配时开始检索字符串的位置。当它找到了与表达式相匹配的文本时，在匹配之后，它把正则表达式的 `lastIndex` 属性设置为下一次匹配执行的第 1 个字符的位置。这就是说，可以通过反复地调用 `exec()`方法来遍历字符串中的所有匹配文本。当 `exec()`再也找不到匹配的文本时，它将返回 `null`，并且把属性 `lastIndex` 重置为 0。

例如，在下面的这个示例中，定义正则表达式直接量，用来匹配字符串 `s` 中的每个字符。在循环结构的条件表达式中反复执行匹配模式，并根据返回结果的值是否为 `null` 作为循环条件。当返回值为 `null` 时，说明字符串检测完毕。然后，读取返回数组 `a` 中包含的匹配子字符串，并调用该数组的属性 `index` 和 `lastIndex`。其中 `index` 显示当前匹配子字符串的起始位置，而 `lastIndex` 属性显示下一次匹配操作的起始位置。代码如下：

```
var s = "javascript";           //测试使用的字符串直接量
var r = /\w/g;                 //匹配模式
while((a = r.exec(s)) != null){ //循环执行匹配操作
    alert(a[0] + "\n" + a.index + "\n" + r.lastIndex); //显示每次匹配操作是返回的结果数组信息
}
```

实际上，通过循环结构反复调用 `exec()`方法是唯一一种获得全局模式的完整模式匹配信息的方法。无论正则表达式是否是全局模式，`exec()`方法都会将完整的细节添加到返回数组中。字符串对象的 `match()`方法就不同，它在全局模式下返回的数组中不会包含这么多的细节信息。

(2) test()方法

`test()`方法要比 `exec()`方法简单，它能够检测参数字符串是否包含至少一个与当前正则表达式的匹配。如果包含匹配就返回 `true`，否则就返回 `false`。例如，在下面示例中，方法 `test()`就会返回 `true`，因为在字符串 `s` 中包含很多个匹配：

```
var s = "javascript";
var r = /\w/g;           //匹配字符
var b = r.test(s);       //返回 true
```

同样使用下面正则表达式也能够匹配，并返回 `true`：

```
var r = /javascript/g;
var b = r.test(s);       //返回 true
```

但是如果使用下面这个正则表达式进行匹配，就会返回 `false`，因为在字符串 `javascript` 中就找不到对应的匹配：

```
var r = /\d/g;           //匹配数字
var b = r.test(s);       //返回 false
```

调用 `test()`方法等价于调用 `exec()`方法。如果 `exec()`方法返回值不是 `null`，则 `test()`方法将返回 `true`。因此，当一个全局正则表达式调用方法 `test()`时，它的行为与 `exec()`方法相同，即它将从 `lastIndex` 属性值指定的位置开始验证字符串。如果发现了匹配，就会将 `lastIndex` 属性值设置为紧邻当前匹配字符串后的字符位置。因此，就可以使用 `test()`方法代替 `exec()`方法来遍历字符串，检测匹配的字符。

例如，针对 `exec()`方法中的循环遍历匹配，也可以这样设计：

```
var s = "javascript";           //测试使用的字符串直接量
var r = /\w/g;                 //匹配模式
while(r.test(s)){              //循环执行匹配验证，如果返回 true，则连续执行验证
    alert(RegExp.lastMatch + "\n" + r.lastIndex); //利用静态属性和实例属性，显示当前匹配信息
}
```

字符串对象支持 `search()`、`replace()`和 `match()`方法实现正则表达式的模式匹配操作，而正则表达式对象支持 `test()`和 `exec()`方法实现模式匹配操作。但是，对于字符串对象的 `search()`、`replace()`和 `match()`方法来说，它们返回的匹配信息显然没有正则表达式的 `test()`和 `exec()`方法详细。

例如，`test()`和 `exec()`方法支持属性 `lastIndex`，而字符串对象的方法只是将 `lastIndex` 属性重置为 0。如果使用一个全局模式的 `exec()`方法或 `test()`方法来检索多个字符串，那么就必须要找到每个字符串中的所有匹配，

以便 `lastIndex` 属性会被自动重置为 0，或者必须明确地将 `lastIndex` 属性设置为 0；否则，当再次检索一个新字符串时，起始位置可能就是原来的字符串中的一个任意位置，而不一定是字符串的开头。

同时，对于 `test()` 和 `exec()` 方法来说，只有声明了 `g` 标志的正则表达式，才会发生这种特殊的 `lastIndex` 属性行为。如果正则表达式对象没有标志 `g`，`exec()` 和 `test()` 将忽略它的 `lastIndex` 属性。

11.2.7 联系表单验证

客户端验证的最大优势就是即时性，这是服务器端验证所无法替代的。服务器端代码，无论是 ASP、PHP，还是其他富有想象力的缩写代码，都需要重载页面才能够生效，除非使用异步交互，但是异步交互仍然需要 JavaScript。通过 jQuery，可以对表单中必填的字段在失去焦点后或者按下键盘键之后，能够立即做出反应，判断用户是否输入信息，否则中止下面的操作，或者提出警示信息。

表单验证的任务可以归纳下面几种类型：

- ☒ 必填检查。
- ☒ 范围校验。
- ☒ 比较验证。
- ☒ 格式验证
- ☒ 特殊验证。

是否必填项验证是最基本的，如输入框获得焦点提示、输入框失去焦点验证错误提示、输入框失去焦点验证正确提示。首先确定输入框是否是必填项，然后就是提示消息的显示位置。

输入参数中的范围校验要稍微复杂一些，因为有输入值的范围。校验做了输入的数据类型为字符串、数字和时间的区分。如果是字符串，则比较字符串的长短，数字和时间则比较大小。

输入参数与另外一个控件值要相对比较简单，因为使用正则表达式，无须自己去思考输入的情况，只需要引入一个正则表达式即可。对于格式验证和特殊任务，都必须通过正则表达式才能够完成。

1. 必填字段验证

以 11.1.1 节示例为基础，在联系表单中当用户按下 Tab 键，或者在输入字段外单击时，JavaScript 能够检查每个必填字段是否为空。为了简化演示，可以为必填字段添加一个统一的标识类 `required`，当必填字段被隐藏后，将移出这些类。

有了这些 `required` 类，就可以在用户没有填写字段时给出提示，这些提示信息被动态添加到对应字段的后面，并添加 `warning` 类，以便设计统一的提示信息类样式。代码如下：

```
if ($(this).is('.required')) {
    var $listItem = $(this).parents('li:first');
    if (this.value == "") {
        var errorMessage = '必须填写';
        if ($(this).is('.conditional')) {
            errorMessage = '当勾选了前面复选框后,' + errorMessage;
        }
        $('</span>')
            .addClass('error-message')
            .text(errorMessage)
            .appendTo($listItem);
        $listItem.addClass('warning');
    }
};
```

上面代码在每个表单输入字段后发生 `blur` 事件时，检测 `required` 类，然后检查空字符串。如果都为 `true`，则提示错误信息，并把这个错误信息添加到父元素 `li` 中。如果要对条件文本字段进行检测，并显示不同的

提示信息, 则可以在对应的复选框被选中后, 显示对应的错误提示信息。演示效果如图 11.9 所示。

在设计这个验证检测脚本时, 应该考虑到当用户取消选中复选框之后, 能够自动取消错误提示信息, 或者当用户再次填写信息时, 能够自动清除这个提示信息。

2. 格式验证

格式验证是表单验证中最常应用的类型, 也是 JavaScript 最擅长的。有时, 如果填写的某个字段文本不正确, 应该友好地进行提示, 以方便用户更有效地使用表单。格式验证包括电子邮件、电话和信用卡等, 当然只要读者掌握了正则表达式的编写, 再复杂的格式都可以通过一行正则表达式来实现。下面就以电子邮件的格式化验证为例进行说明, 代码如下:

```
if ($(this).is('#email')) {
    var $listItem = $(this).parents('li:first');
    if (this.value != "" && !/^[a-zA-Z]{2,4}$/.test(this.value)) {
        var errorMessage = '电子邮件格式不正确';
        $('<span></span>')
            .addClass('error-message')
            .text(errorMessage)
            .appendTo($listItem);
        $listItem.addClass('warning');
    }
};
```

在上面代码中, 首先检测电子邮件字段, 然后把父列表项保存到一个变量中, 再用两个条件检测该值是否为空, 以及是否匹配正则表达式。如果检测成功, 创建一个错误信息, 将这条信息插入到标签, 并把错误信息和标签添加到父列表项中, 同时为父列表项添加 warning 类。

对于正则表达式验证, 这里使用了 test() 方法, 而不是 replace() 方法, 因为这里仅需要检测是否匹配, 即检测返回值是否为 true 或者 false。对于正则表达式的设计, 为了使检测更精确, 需要查找电子邮件中的 @ 和 . 这两个特殊字符标识, 以及 . 字符后面应该有 2~4 字符来表示域名扩展符。演示效果如图 11.10 所示。

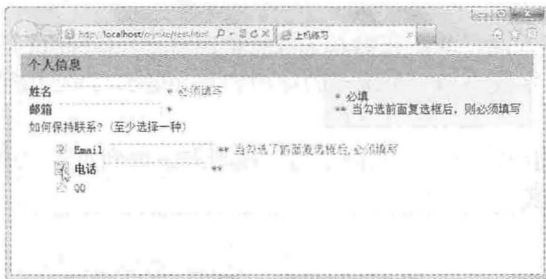


图 11.9 检测必填字段

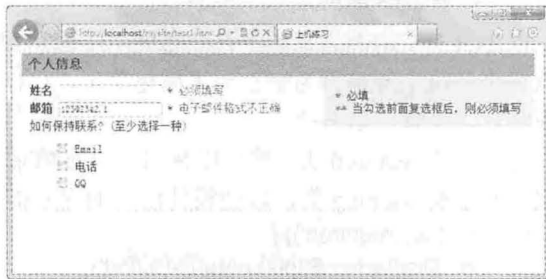


图 11.10 检测格式

考虑到每次验证时用户可能补写信息, 此时代码应该即时擦除对应的错误提示信息, 因此不妨在这两段代码前面对错误提示信息进行清除, 避免一旦出现了错误信息, 就一直显示在那儿。代码如下:

```
$('form :input').blur(function() {
    $(this).parents('li:first').removeClass('warning')
    .find('span.error-message').remove();
    if ($(this).is('.required')) {
        var $listItem = $(this).parents('li:first');
        if (this.value == "") {
            var errorMessage = '必须填写';
            if ($(this).is('.conditional')) {
                errorMessage = '当勾选了前面复选框后,' + errorMessage;
            }
        }
    }
});
```

```

    $('<span></span>')
      .addClass('error-message')
      .text(errorMessage)
      .appendTo($listItem);
    $listItem.addClass('warning');
  };
};
if ($(this).is('#email')) {
  var $listItem = $(this).parents('li:first');
  if (this.value != "" && !/.+@.+[a-zA-Z]{2,4}$/.test(this.value)) {
    var errorMessage = '电子邮件格式不正确';
    $('<span></span>')
      .addClass('error-message')
      .text(errorMessage)
      .appendTo($listItem);
    $listItem.addClass('warning');
  };
};
});

```

3. 综合检测

在上面检测中，都是基于用户把焦点置于对应文本框之中，然后移开之后发生的。但是如果用户根本就没有接触这些字段，而直接提交表单，那么就会发生很多问题。因此，有必要在用户提交表单时，对整个表单的信息进行一次综合检测，以防止错填或者漏填的问题发生。

通过表单的 submit() 事件处理函数，可以在所有的必填字段上触发 blur 事件。首先在这个处理函数中移除不存在的元素，然后在后面再动态添加，因为这些信息都是动态显示的。在触发 blur 事件之后，获取当前表单中包含的 warning 类的总数。如果存在 warning 类，就创建一个新的 id 为 submit-message 的 <div> 的标签，并把它插入到提交按钮的前面，方便用户阅读，最后还要阻止表单提交。代码如下：

```

$('form').submit(function() {
  $('#submit-message').remove();
  $(':input.required').trigger('blur');
  var numWarnings = $('<div>').length;
  if (numWarnings) {
    var fieldList = [];
    $('<div>').each(function() {
      fieldList.push($(this).text());
    });
    $('<div>')
      .attr({
        'id': 'submit-message',
        'class': 'warning'
      })
      .append('请重新填写下面 ' + numWarnings + ' 个字段:<br />')
      .append('&bull; ' + fieldList.join('<br />&bull; '))
      .insertBefore('#send');
    return false;
  };
});

```

在上面代码中，首先定义一个空数组 fieldList，然后去掉每个带 warning 类的元素的后代 <label> 标签，将该标签中的文本使用 JavaScript 本地 push 函数推到 fieldList 数组中，这样每个标签中的文本就构成了

fieldList 数组中的一个独立元素。然后,修改 submit-message 元素,将 fieldList 数组中的内容添加到这个<div>标签中,并使 JavaScript 本地函数 join()将数组转换为字符串,将每个数组元素与一个换行符和一个圆点符号连接在一起。这个 HTML 标记只是显示而不具有语义性,而且可以随时废弃,因此不需要过分考虑动态信息的语义结构问题。最后演示效果如图 11.11 所示。

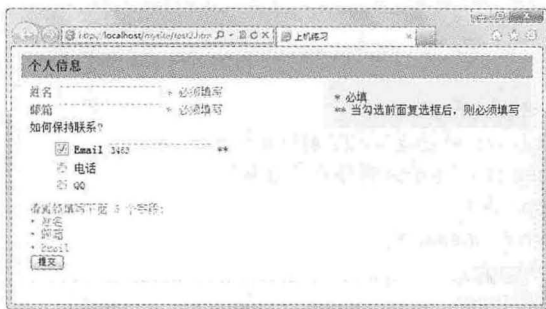


图 11.11 综合检测

11.3 增强型表单

表单包含表单域、输入框、下拉框、单选框、多选框和按钮等元素,每个元素在表单中所起到的作用也不相同,而且每个元素都有各自的特性和用途。但是在实际应用中,这些表单对象还无法满足设计需求,用户需要借助 JavaScript 改善和增强它们的功能,以提高可用性。

11.3.1 自适应多行文本框

自适应多行文本框功能开始于谷歌翻译文本框,如图 11.12 所示。当用户在文本框中输入任意多行语句时,文本框会自动根据文本内容的高度进行调节,以实现自动包容这些文本,而不是显示滚动条。

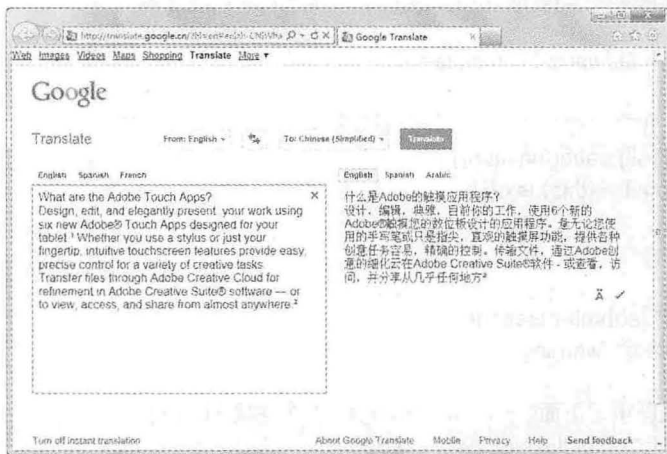


图 11.12 谷歌翻译的自适应多行文本框

下面利用 jQuery 扩展一个插件 Elastic,使用这个插件实现 Textarea 文本框输入的数据自动改变输入框大小。演示效果如图 11.13 所示。

Elastic 插件的设计思路为:先定义一个临时的 div 元素,并把文本框中的多行文本转换为 HTML 内容。这个内容传递给隐藏的临时 div 元素后,获取这个临时元素的高度,利用这个高度重新设计文本框的高度,

从而实现文本框高度自适应的设计功能。

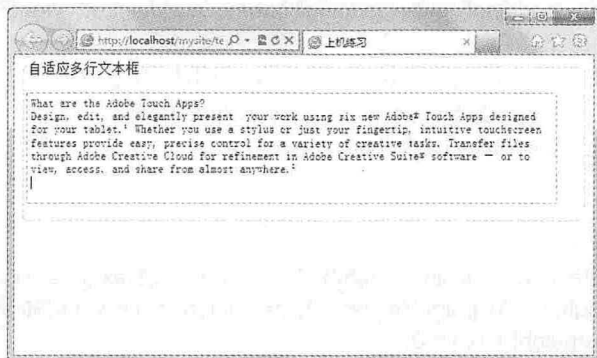


图 11.13 定义自适应的文本框

首先定义一个插件命令接口函数。代码如下：

```
(function(jQuery) {
    jQuery.fn.extend({
        elastic : function() {
            //准备编写的代码
        }
    });
})(jQuery);
```

在 elastic()方法中，定义两个私有函数，其中第 1 个函数 setHeightAndOverflow()用来动态设置文本框的高度和显示方式。代码如下：

```
function setHeightAndOverflow(height, overflow) {
    curratedHeight = Math.floor(parseInt(height, 10));
    if($textarea.height() != curratedHeight) {
        $textarea.css({
            'height' : curratedHeight + 'px',
            'overflow' : overflow
        });
    }
}
```

第 2 个私有函数 update()能够利用临时隐藏元素 div 的高度以及其他关键样式信息，来设置文本框的对应样式，特别是高度值。代码如下：

```
function update() {
    var textareaContent = $textarea.val().replace(/&/g, '&amp;').replace(/</g, '<');
    var twinContent = $twin.html();
    if(textareaContent + '&nbsp;' != twinContent) {
        $twin.html(textareaContent + '&nbsp;');
        if(Math.abs($twin.height() + lineHeight - $textarea.height()) > 3) {
            var goalheight = $twin.height() + lineHeight;
            if(goalheight >= maxheight) {setHeightAndOverflow(maxheight, 'auto');}
            else if(goalheight <= minheight) {setHeightAndOverflow(minheight, 'hidden');}
            else {setHeightAndOverflow(goalheight, 'hidden');}
        }
    }
}
```

在 elastic()方法内部，首先定义一个样式列表数组，该数组存储着将要把文本框的对应样式和属性传递

给临时隐藏元素 div 的列表。代码如下：

```
var mimics = ['paddingTop', 'paddingRight', 'paddingBottom', 'paddingLeft', 'fontSize', 'lineHeight', 'fontFamily', 'width', 'fontWeight'];
```

然后新建一个临时的 div 元素，以绝对定位方式显示，初始化为隐藏显示，再把这个 div 元素添加到文本框的父元素内部的尾部。分别读入文本框的行高、最小高度等信息。

```
var $textarea = jQuery(this), $twin = jQuery('<div />').css({
    'position' : 'absolute',
    'display' : 'none',
    'word-wrap' : 'break-word'
}), lineHeight = parseInt($textarea.css('line-height'), 10) || parseInt($textarea.css('font-size'), '10'), minHeight =
parseInt($textarea.css('height'), 10) || lineHeight * 3, maxHeight = parseInt($textarea.css('max-height'), 10) ||
Number.MAX_VALUE, goalheight = 0, i = 0;
if(maxHeight < 0) {
    maxHeight = Number.MAX_VALUE;
}
$twin.appendTo($textarea.parent());
```

接着，利用 mimics 数组信息，根据数组中存储的样式属性，逐一把文本框的对应样式属性值传递给 div 元素，以便模拟相等的外形尺寸。代码如下：

```
var i = mimics.length;
while(i--){
    $twin.css(mimics[i].toString(), $textarea.css(mimics[i].toString()));
}
```

最后，禁止文本框显示滚动条，并在键盘抬起和粘贴文本时，初始化状态分别调用 update() 函数，实现文本框的高度自适应变化，以在不显示滚动条的情况下包容所有的文本。代码如下：

```
$textarea.css({
    'overflow' : 'hidden'
});
$textarea.keyup(function() {update();
});
$textarea.live('input paste', function(e) {setTimeout(update, 250);
});
update();
```

设计完 elastic() 插件之后，在页面初始化事情处理函数中为 jQuery 匹配的文本框调用 elastic() 方法即可。演示效果如图 11.11 所示。

```
$(function(){
    $('textarea').elastic();
})
```

11.3.2 注册码文本框

注册码文本框功能源于传统桌面应用程序的输入注册码界面，在很多软件安装界面中都可以看到这样的文本框。多个文本框能够捆绑在一起，允许用户输入特定格式的注册码，当输入一个文本框之后，输入焦点会自动跳转到下一个文本框，依此类推。如图 11.14 所示是 Adobe CS5 套装软件的安装界面之一，该界面的序列号输入框就是一个典型的注册码文本框功能应用。

下面就来模仿这种文本框输入功能，它提供了自动跳格功能，当用户输入的字符数一旦超出已定义的最大长度，则会根据事先设置的目标自动跳转到相应元素上，省却了用户按 Tab 键的麻烦。同样，如果按退格键，可以连续清理多个跳格文本框的内容，按 Esc 键可以快速清理关联文本框中所有内容。另外，本案例还提供了对文本框输入内容进行过滤的功能。演示效果如图 11.15 所示。

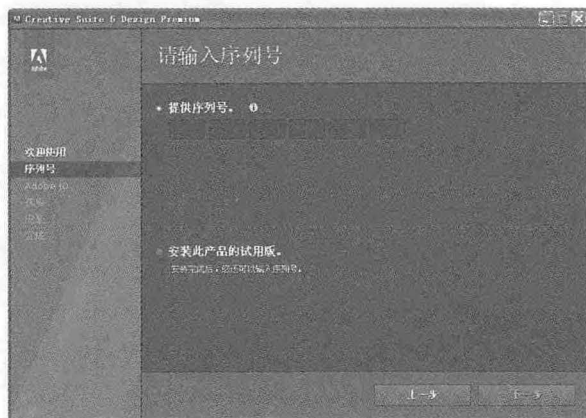


图 11.14 Adobe CS5 套装软件的安装界面

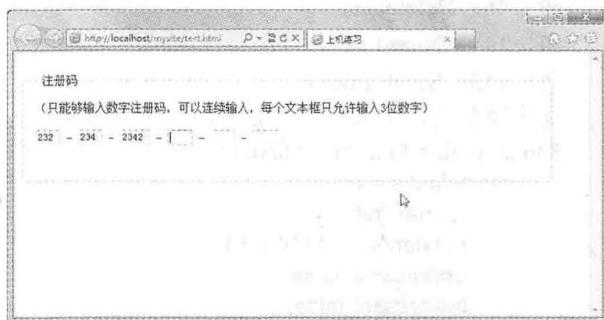


图 11.15 注册码文本框

首先, 定义一个封闭的命名空间, 把 jQuery 传递给它。代码如下:

```
(function($) {
    //开发区域
})(jQuery);
```

在该命名空间中为 jQuery 扩展一个命令 `autotab()`, 该命令用来设置自动 Tab 机制的元素和输入文本框的格式参数。可设格式选项如下。

- ☒ `format`: 字符格式。
- ☒ `maxlength`: 最大长度。
- ☒ `uppercase`: 大写。
- ☒ `lowercase`: 小写。
- ☒ `nospace`: 空格符。
- ☒ `target`: 目标。
- ☒ `previous`: 前一个对象。
- ☒ `pattern`: 模式。

然后分别对这些参数进行检测和预处理, 其中 `key` 变量存储一些特殊键。说明如下。

```
var keys = [8, 9, 16, 17, 18, 19, 20, 27, 33, 34, 35, 36, 37, 38, 39, 40, 45, 46, 144, 145];
```

- ☒ 8: Backspace。
- ☒ 9: Tab。
- ☒ 16: Shift。
- ☒ 17: Ctrl。
- ☒ 18: Alt。
- ☒ 19: Pause Break。
- ☒ 20: Caps Lock。
- ☒ 27: Esc。
- ☒ 33: Page Up。
- ☒ 34: Page Down。
- ☒ 35: End。
- ☒ 36: Home。
- ☒ 37: Left Arrow。
- ☒ 38: Up Arrow。
- ☒ 39: Right Arrow。

- ☑ 40: Down Arrow。
- ☑ 45: Insert。
- ☑ 46: Delete。
- ☑ 144: Num Lock。
- ☑ 145: Scroll Lock。

这些键在输入文本框中将被屏蔽。代码如下：

```
$.fn.autotab = function(options) {
    var defaults = {
        format: 'all',
        maxlength: 2147483647,
        uppercase: false,
        lowercase: false,
        nospace: false,
        target: null,
        previous: null,
        pattern: null
    };
    $.extend(defaults, options);
    if(typeof defaults.target == 'string')
        defaults.target = check_element(defaults.target);
    if(typeof defaults.previous == 'string')
        defaults.previous = check_element(defaults.previous);
    var maxlength = $(this).attr('maxlength');
    if(defaults.maxlength == 2147483647 && maxlength != 2147483647)
        defaults.maxlength = maxlength;
    else if(defaults.maxlength > 0)
        $(this).attr('maxlength', defaults.maxlength)
    else
        defaults.target = null;
    if(defaults.format != 'all')
        $(this).autotab_filter(defaults);
    return $(this).bind('keydown', function(e) {
        if(e.which == 8 && this.value.length == 0 && defaults.previous)
            defaults.previous.focus().val(defaults.previous.val());
    }).bind('keyup', function(e) {
        var keys = [8, 9, 16, 17, 18, 19, 20, 27, 33, 34, 35, 36, 37, 38, 39, 40, 45, 46, 144, 145];
        if(e.which != 8) {
            var val = $(this).val();
            if($.inArray(e.which, keys) == -1 && val.length == defaults.maxlength && defaults.target)
                defaults.target.focus();
        }
    });
};
```

利用这个命令可以设置输入字符的格式选项，如`$('#phone').autotab({ format: 'number' })`表示只能够输入数字；`$('#username').autotab({ format: 'alphanumeric', target: 'password' })`表示只能够输入字符和数字，且目标域为 password；`$('#password').autotab({ previous: 'username', target: 'confirm' })`表示设置前一个文本框为用户名文本框等。

再定义 `autotab_filter()` 命令，该方法负责数据过滤和加工，过滤条件根据参数设置而定。数据格式参数项（format）包括以下几种选项值。

- ☑ text: 数字文本。
- ☑ alpha: 字符。

- ☑ number: 数字。
- ☑ numeric: 数字。
- ☑ alphanumeric: 数字和字符。
- ☑ custom: 自定义, 将根据 pattern 参数定义的正则表达式确定。
- ☑ all: 任意数字。
- ☑ default: 任意数字。

如果设置了 nospace 参数项, 将清除输入的空字符。

如果设置了 uppercase 参数项, 将输入的字符转换为大写形式。

如果设置了 lowercase 参数项, 将输入的字符转换为小写形式。

```
$.fn.autotab_filter = function(options) {
    var defaults = {
        format: 'all',
        uppercase: false,
        lowercase: false,
        nospace: false,
        pattern: null
    };
    if(typeof options == 'string' || typeof options == 'function')
        defaults.format = options;
    else
        $.extend(defaults, options);
    for(var i = 0; i < this.length; i++){
        $(this[i]).bind('keyup', function(e) {
            var val = this.value;
            switch(defaults.format){
                case 'text':
                    var pattern = new RegExp('[0-9]+', 'g');
                    val = val.replace(pattern, "");
                    break;
                case 'alpha':
                    var pattern = new RegExp('[^a-zA-Z]+', 'g');
                    val = val.replace(pattern, "");
                    break;
                case 'number':
                case 'numeric':
                    var pattern = new RegExp('[^0-9]+', 'g');
                    val = val.replace(pattern, "");
                    break;
                case 'alphanumeric':
                    var pattern = new RegExp('[^0-9a-zA-Z]+', 'g');
                    val = val.replace(pattern, "");
                    break;
                case 'custom':
                    var pattern = new RegExp(defaults.pattern, 'g');
                    val = val.replace(pattern, "");
                    break;
                case 'all':
                default:
                    if(typeof defaults.format == 'function')
                        var val = defaults.format(val);
                    break;
            }
        });
    }
    if(defaults.nospace)
```

```

    {
        var pattern = new RegExp('[ ]+', 'g');
        val = val.replace(pattern, "");
    }
    if(defaults.uppercase)
        val = val.toUpperCase();
    if(defaults.lowercase)
        val = val.toLowerCase();
    if(val != this.value)
        this.value = val;
    });
}
};

```

该方法能够根据参数指定的格式处理输入的文本。参数对象可以包含 `format`、`uppercase`、`lowercase`、`nospace`、`pattern` 几个选项。其中，`format` 参数项可以设置输入文本的格式，取值包括 `text`、`number`、`alphanumeric`、`all`、`custom`；`uppercase` 参数项可以定义大写格式，取值包括 `true` 和 `false`，默认值为 `false`；`lowercase` 参数项可以定义小写格式，取值包括 `true` 和 `false`，默认值为 `false`；`nospace` 参数项可以定义过滤空字符，取值包括 `true` 和 `false`，默认值为 `false`；`pattern` 参数项可以设置自定义模式，以正则表达式字符串形式传递，默认为 `null`。例如，`$('#number1, #number2, #number3').autotab_filter('number')`、`$('#product_key').autotab_filter({ format: 'alphanumeric', nospace: true })`、`$('#unique_id').autotab_filter({ format: 'custom', pattern: '^[^0-9\\.]' })`。

最后定义一个 `autotab_magic()` 命令，该方法能够自动设置输入文本框的焦点，当输入完第 1 个文本框后，输入焦点会自动跳转到下一个输入文本框；如果删除当前文本框的所有字符，则控制焦点会自动跳转到前一个输入文本框的最后一个字符后。

```

$.fn.autotab_magic = function(focus) {
    for(var i = 0; i < this.length; i++){
        var n = i + 1;
        var p = i - 1;
        if(i > 0 && n < this.length)
            $(this[i]).autotab({ target: $(this[n]), previous: $(this[p]) });
        else if(i > 0)
            $(this[i]).autotab({ previous: $(this[p]) });
        else
            $(this[i]).autotab({ target: $(this[n]) });
        if(focus != null && (isNaN(focus) && focus == $(this[i]).attr('id')) || (!isNaN(focus) && focus == i))
            $(this[i]).focus();
    }
    return this;
};

```

完成 jQuery 的 3 个扩展命令之后，在文档中构建一个多文本框的注册码模拟界面。代码如下：

```

<form id="contact" action="index.html" method="get">
    <fieldset>
        <legend>注册码</legend>
        <p>（只能输入数字注册码，可以连续输入，每个文本框只允许输入 3 位数字）</p>
        <p id="info">
            <input type="text" name="num1" id="num1" maxlength="3" size="3" />
            -
            <input type="text" name="num2" id="num2" maxlength="3" size="3" />
            -
            <input type="text" name="num3" id="num3" maxlength="4" size="4" />
            -
            <input type="text" name="num4" id="num4" maxlength="3" size="3" />
        </p>
    </fieldset>
</form>

```

```

-
<input type="text" name="num5" id="num5" maxlength="3" size="3" />
-
<input type="text" name="num6" id="num6" maxlength="4" size="4" />
</p>
</fieldset>
</form>

```

最后在页面初始化事件处理函数中为这些文本框绑定插件命令即可。代码如下：

```

$(function(){
    $('#info :input').autotab_magic().autotab_filter('numeric');
})

```

11.3.3 掩码输入文本框

表单填写过程中，经常需要输入日期、电话号码、邮编、限位字符等带有格式的项目，这些复杂的格式如果没有提示性的说明，很容易输入错误。虽然可以通过表单验证的方式来纠正用户的输入信息，但是这种纠正都是发生在输入完毕之后，从而导致用户为了能够填写符合要求的格式内容，往往要花费大量的时间，这为表单的使用带来了障碍，也不符合表单使用易用性的设计宗旨。

本节将借助 jQuery 开发一个掩码输入器，帮助用户解决这个问题。该掩码输入器在使用时，需要与输入文本框进行绑定，并设置好文本框的输入掩码格式，这样当用户在执行表单填写时，文本框会自动显示对应的格式状态，只有符合输入要求和格式的内容才有效，允许继续执行。演示效果如图 11.16 所示。

本案例的掩码输入器允许用户更加容易地根据固定宽度输入特定格式的数据，如电话、日期、邮编等。构建输入文本框后，把 maskedInput 插件绑定到该文本框上，并为其传递一个特定格式的数据，如 99/99/9999、(999) 999-9999。在这个特定格式数据中，字符的含义说明如下。

- ☑ a：代表一个字母字符，包括 A~Z 和 a~z。
- ☑ 9：代表一个数字字符，包括 0~9。
- ☑ *：表示一个字母或数字字符，包括 A~Z、a~z 和 0~9。

当然也可以自定义掩码格式，自定义限制用户的输入范围。例如，电话区号一般以 0 开头，而月份的数字不能够大于 13。为了避免用户随意输入数字，可以自定义一个格式模式，限制用户在输入月份时第 1 个数字必须是 0 或者 1，而区号的第 1 个数字必须是 0。实现的代码如下：

```

<script type="text/javascript">
$(function(){
    $.mask.definitions['0']='[0]';           //设置数字 0 的模式为必须输入数字 0
    $.mask.definitions['1']='[01]';         //设置数字 1 的模式为必须输入数字 0 或者 1
    $('#date').mask('19/99/9999');          //定义第 1 个数字为模式 1
    $('#phone').mask('(099) 999-9999');     //定义第 1 个数字为模式 0
})
</script>

```

如果不喜欢下划线作为掩码（_）字符，可以传递一个可选的参数给 maskedInput 的 mask() 方法。例如，在下面脚本中将定义 ? 为掩码字符：

```

<script type="text/javascript">
$(function(){

```



图 11.16 掩码输入框


```

$('#date').mask('99/99/9999');
$('#phone').mask('(999) 999-9999', {
    placeholder:"?" //定义?为掩码字符
});
})
</script>

```

下面就来分解该插件的实现过程。

首先，在一个独立的命名空间中为 jQuery 绑定一个插件 mask()，并为 jQuery 定义一个 mask 对象，该对象包含一个 definitions 属性，利用该属性可以自定义掩码格式。默认情况下，插件定义了 3 个格式，其中 \$.mask.definitions.9 表示数字，\$.mask.definitions.a 表示字符，而 \$.mask.definitions.* 表示数字或者字符。代码如下：

```

(function($) {
    $.mask = {
        definitions: {
            '9': "[0-9]",
            'a': "[A-Za-z]",
            '*': "[A-Za-z0-9]"
        }
    };
    $.fn.extend({
        mask: function(mask, settings) {
        }
    });
})(jQuery);

```

初始化变量，检测 mask 参数值是否为空，如果为空则返回。然后初始化参数对象，根据用户设置的 settings 参数更新插件中默认的 settings 参数值。代码如下：

```

if (!mask && this.length > 0) {
    var input = $(this[0]);
    var tests = input.data("tests");
    return $.map(input.data("buffer"), function(c, i) {
        return tests[i] ? c : null;
    }).join("");
}
settings = $.extend({
    placeholder: "_",
    completed: null
}, settings);
var defs = $.mask.definitions;
var tests = [];
var partialPosition = mask.length;
var firstNonMaskPos = null;
var len = mask.length;

```

把掩码参数值劈开为数组，并逐个进行分析。如果掩码字符为?，则忽略；如果与 \$.mask.definitions 掩码列表匹配，则代入其定义的正则表达式，然后构建一个新的正则表达式对象，并它推入到 tests 数组中；如果是其他字符，则直接向 tests 数组中推入一个 null 值。代码如下：

```

$.each(mask.split(""), function(i, c) {
    if (c == '?') {
        len--;
        partialPosition = i;
    } else if (defs[c]) {
        tests.push(new RegExp(defs[c]));
        if(firstNonMaskPos==null)

```

```

        firstNonMaskPos = tests.length - 1;
    } else {
        tests.push(null);
    }
});

```

完成了初始化工作，下面就开始调用 `each()` 方法，逐一为每个输入文本框设计掩码格式。同时为当前文本框绑定焦点事件、按下键盘和输入字符等事件，以便动态跟踪用户输入状态，以及文本框中被输入的字符。代码如下：

```

return this.each(function() {
    var input = $(this); //获取当前文本框
    //把掩码转换为数组，并通过映射进行处理
    var buffer = $.map(mask.split(""), function(c, i) { if (c != '?') return defs[c] ? settings.placeholder : c });
    var ignore = false;
    var focusText = input.val();
    input.data("buffer", buffer).data("tests", tests); //定义缓存数据
    if (!input.attr("readonly")) //如果文本框可以编辑，则先清除先前的掩码，然后重新设置掩码
        input
            .one("unmask", function() {
                input
                    .unbind(".mask")
                    .removeData("buffer")
                    .removeData("tests");
            })
            .bind("focus.mask", function() {
                focusText = input.val();
                var pos = checkVal();
                writeBuffer();
                setTimeout(function() {
                    if (pos == mask.length)
                        input.caret(0, pos);
                    else
                        input.caret(pos);
                }, 0);
            })
            .bind("blur.mask", function() {
                checkVal();
                if (input.val() != focusText)
                    input.change();
            })
            .bind("keydown.mask", keydownEvent)
            .bind("keypress.mask", keypressEvent)
            .bind("pasteEventName", function() {
                setTimeout(function() { input.caret(checkVal(true)); }, 0);
            });
    checkVal();
});
}

```

在 `this.each()` 方法的参数回调函数中定义几个私有函数，其中 `shiftL()` 和 `shiftR()` 能够根据掩码当前的光标下标位置执行向前和向后操作，以方便用户在输入字符、删除字符后能够自动用输入的字符替换掩码字符，或者把输入的字符替换为掩码字符。代码如下：

```

function shiftL(pos) {
    while (!tests[pos] && --pos >= 0);
    for (var i = pos; i < len; i++) {

```

```

        if (tests[i]) {
            buffer[i] = settings.placeholder;
            var j = seekNext(i);
            if (j < len && tests[j].test(buffer[j])) {
                buffer[i] = buffer[j];
            } else
                break;
        }
    }
    writeBuffer();
    input.caret(Math.max(firstNonMaskPos, pos));
};

function shiftR(pos) {
    for (var i = pos, c = settings.placeholder; i < len; i++) {
        if (tests[i]) {
            var j = seekNext(i);
            var t = buffer[i];
            buffer[i] = c;
            if (j < len && tests[j].test(t))
                c = t;
            else
                break;
        }
    }
};

```

keydownEvent()和 keypressEvent()内部函数定义了两个事件侦测和处理行为，分别用来侦测用户键盘输入动作和输入字符后的处理操作。当用户按下键盘时，触发 keydownEvent()事件函数，该函数将检测用户按下的键，并与掩码模板定义的字符要求是否一致。当输入完毕字符之后，触发 keypressEvent()函数，该函数用符合要求的输入字符替换掉掩码字符。代码如下：

```

function keydownEvent(e) {
    var pos = $(this).caret();
    var k = e.keyCode;
    ignore = (k < 16 || (k > 16 && k < 32) || (k > 32 && k < 41));
    if ((pos.begin - pos.end) != 0 && (!ignore || k == 8 || k == 46))
        clearBuffer(pos.begin, pos.end);
    if (k == 8 || k == 46 || (iPhone && k == 127)) { //backspace/delete
        shiftL(pos.begin + (k == 46 ? 0 : -1));
        return false;
    } else if (k == 27) { //escape
        input.val(focusText);
        input.caret(0, checkVal());
        return false;
    }
};

function keypressEvent(e) {
    if (ignore) {
        ignore = false;
        return (e.keyCode == 8) ? false : null;
    }
    e = e || window.event;
    var k = e.charCode || e.keyCode || e.which;
    var pos = $(this).caret();
}

```

```

if (e.ctrlKey || e.altKey || e.metaKey) { // ignore
    return true;
} else if ((k >= 32 && k <= 125) || k > 186) { // typeable characters
    var p = seekNext(pos.begin - 1);
    if (p < len) {
        var c = String.fromCharCode(k);
        if (tests[p].test(c)) {
            shiftR(p);
            buffer[p] = c;
            writeBuffer();
            var next = seekNext(p);
            $(this).caret(next);
            if (settings.completed && next == len)
                settings.completed.call(input);
        }
    }
}
return false;
};

```

最后，还需要定义一个辅助验证函数 `checkVal()`，该函数主要负责对输入的每个字符进行检查。如果输入字符合法，则填写对应位置的文本框值，实现对掩码占位符进行替换，同时把光标移到下一个字符位置。代码如下：


```

function checkVal(allow) {
    var test = input.val();
    var lastMatch = -1;
    for (var i = 0, pos = 0; i < len; i++) {
        if (tests[i]) {
            buffer[i] = settings.placeholder;
            while (pos++ < test.length) {
                var c = test.charAt(pos - 1);
                if (tests[i].test(c)) {
                    buffer[i] = c;
                    lastMatch = i;
                    break;
                }
            }
            if (pos > test.length)
                break;
        } else if (buffer[i] == test[pos] && i != partialPosition) {
            pos++;
            lastMatch = i;
        }
    }
    if (!allow && lastMatch + 1 < partialPosition) {
        input.val("");
        clearBuffer(0, len);
    } else if (allow || lastMatch + 1 >= partialPosition) {
        writeBuffer();
        if (!allow) input.val(input.val().substring(0, lastMatch + 1));
    }
    return (partialPosition ? i : firstNonMaskPos);
};

```


第12章

jQuery UI 开发概述

( 视频讲解：49 分钟)

jQuery 已经被广泛使用，凭借其简洁的 API，对 DOM 强大的操控性，易扩展性越来越受到 Web 开发人员的喜爱。以 jQuery 为基础的 JavaScript 开源的网页用户界面代码库也越来越多。这些应用库包含底层用户交互、动画、特效和可更换主题的可视控件。直接使用这些代码库可以构建具有很好交互性的 Web 应用程序。

本章将讲解 jQuery 用户代码库的开发和使用基础，以便帮助读者学会开发属于自己的代码库，或者灵活使用网上开源的第三方代码库。

12.1 jQuery UI 开发

UI 是 User Interface 一词的简称，翻译为用户界面，也称人机界面，是指用户与某些系统进行交互的方法集合。这些系统不仅是电脑程序，也包括某种特定的机器、设备、复杂的工具等。

jQuery UI 实际上就是 jQuery 插件，不过它侧重于应用，而不是简单地增强 jQuery 功能。jQuery UI 与 jQuery 的主要区别如下：

- ☑ jQuery 是一个 JavaScript 代码库，主要提供的功能是选择器、属性修改和事件绑定等基础性操作。
- ☑ jQuery UI 则是在 jQuery 的基础上，利用 jQuery 的扩展性设计的插件，提供了一些常用的界面元素，如对话框、拖动行为、改变大小行为等。
- ☑ jQuery 本身注重于底层代码逻辑，没有漂亮的界面。而 jQuery UI 则补充了前者的不足，提供了华丽的展示界面，使人更容易接受。它既有强大的底层逻辑，又有华丽的样式和特效。

jQuery UI 主要分为如下 3 个部分。

- ☑ 交互：主要是一些与鼠标交互相关的内容，如 Draggable、Droppable、Resizable、Selectable 和 Sortable 等。
- ☑ 部件：主要是一些界面的扩展，如 Accordion、AutoComplete、ColorPicker、Dialog、Slider、Tabs、DatePicker、Magnifier、ProgressBar、Spinner 等。新版本的 UI 会包含更多的部件。
- ☑ 效果库：用于提供丰富的动画效果，让动画不再局限于 jQuery 的 animate() 方法。

12.1.1 设计思想

一个优秀的 UI 插件需要从样式和脚本两个方面下功夫，要求样式美观、精致，可定制；脚本高效、优

化,可扩展。

很多初学者会认为样式是个很复杂的东西,需要沉着冷静的心态,同时还需要非凡的审美能力,才能设计出赏心悦目的 UI。其实 CSS 也就是那么些属性: position、margin、padding、width、height、left、top、float、border、background 等,UI 设计的漂亮与否在很大程度上依赖于设计人员对配色的把握和整体效果的协调。

【示例 1】在页面中输出显示下面一行文字,并定制文本字体和大小。即便这是个很简单的操作,但是对于不同设计思想的人来说,会得到不同设计效果。

jQuery 设计宗旨: Write Less, Do More。倡导写更少的代码,做更多的事情。

随意者可能会这样设计:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>上机练习</title>
</head>
<body>
<p>jQuery 设计宗旨: Write Less, Do More。倡导写更少的代码,做更多的事情。</p>
</body>
</html>
```

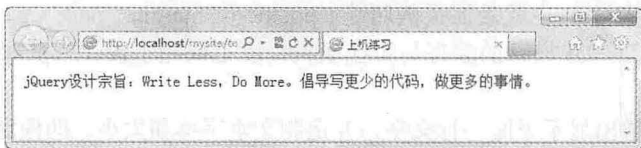
认真者也可以这样设计:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>上机练习</title>
<style type="text/css">
p{ font-family:'宋体'; font-size:12px; }
</style>
</head>
<body>
<p>jQuery 设计宗旨: Write Less, Do More。倡导写更少的代码,做更多的事情。</p>
</body>
</html>
```

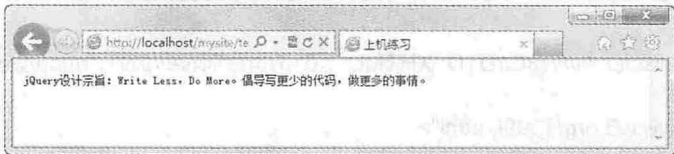
卓越者会这样设计:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>上机练习</title>
<style type="text/css">
p{ font-family:'Verdana','宋体'; font-size:12px; }
</style>
</head>
<body>
<p>jQuery 设计宗旨: Write Less, Do More。倡导写更少的代码,做更多的事情。</p>
</body>
</html>
```

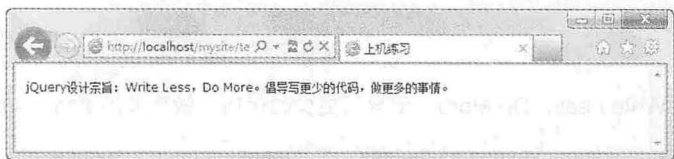
3 位设计者无非在字体样式设计上略有不同,其他代码都一模一样,但是就是这样很小的细节区分,导致 UI 的视觉效果差异明显。这种设计态度的不同,导致设计效果也迥然不同。比较效果如图 12.1 所示。



(a) 没有样式支持



(b) 不考虑视觉效果样式



(c) 考虑视觉效果样式

图 12.1 设计态度决定设计效果

实际上,很多的设计失去关注,正是因为这些不起眼的 font-family 和 font-size 样式。当然这只是一个简单的示例,以此警醒读者掌握 CSS 应该从简单做起,从基本入手,在实践中运用并不断深入。

除了这些外在的效果,需要读者灵活掌握和应用 CSS+HTML,更需要设计人员对 JavaScript 有着深刻的理解。也许部分读者会误认为对 DOM 的操作只需要通过 getElementById()和 getElementsByTagName()基本方法,以及其他的 API 就可以轻松地完成,但是当思路确定后,思想才是重点,一段代码是精华还是糟粕很容易就可以区分出来,究其原因还是取决设计思想。

【示例 2】 同样都是设计一个简单的列表结构,希望通过脚本形式在文档中输出 10 个列表项,列表项的内容分别为数组元素的值,同时分别为每项列表绑定一个 class 属性值,演示效果如图 12.2 所示。不同的脚本,反应了设计师设计思想的差异。



图 12.2 动态输出列表结构

方法一:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var a = new Array(1,2,3,4,5,6,7,8,9,0);
    var menu = "";
    for (var i = 0; i < a.length; i++) {
        menu += '<li class="style_' + a[i] + '">' + a[i] + '</li>';
    }
    $("ul").html(menu);
})
</script>
<title>上机练习</title>
</head>
<body>
<ul></ul>
</body>
</html>

```

方法二:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script type="text/javascript" >
String.prototype.format = function() {
    var args = arguments;
    return this.replace(/\{\d+\}/g, function() {
        return args[arguments[1]];
    });
};
$(function(){
    var a = new Array(1,2,3,4,5,6,7,8,9,0);
    var m = '<li class="style_{0}">{0}</li>';
    var menu = "";
    for (var i = 0; i < a.length; i++) {
        menu += m.format(a[i]);
    }
    $("ul").html(menu);
})
</script>
<title>上机练习</title>
</head>
<body>
<ul></ul>
</body>
</html>

```

显然，第 2 种方法的代码显得更加优雅，更加高效，这种代码读起来显然更具吸引力，它具有稳健且高效的可扩展性。这里不仅仅是代码的差异，而是一种设计思想的区别。谈及设计思想，不是简单地复制

和拷贝所能够达到的，需要读者具有扎实的编程基本功，同时还应该拥有良好的设计思路以及高亢的设计热情。

12.1.2 设计体验

jQuery 开发或使用更多的灵感是来自实践，而不是拿来主义。下面通过一个案例讲解 jQuery UI 开发过程，以便更形象地引导读者掌握 UI 设计思想和操作步骤。

1. 设计目标

在 UI 开发之前，用户应该明确设计目标，对设计目标有一个清醒的认识，有很明确的方向感。本案例的设计目标就是呈现一个用于 UI 的 Slider（滑动条）。滑动条有以下两个作用：

- ☑ 方便使用鼠标控制对象大小，或者通过鼠标拖曳改变变量的值。
- ☑ 动态显示进程的状态，如加载进度等。

本案例演示效果如图 12.3 所示。

2. 设计草图

动手编码之前，需要画一个草图，简单描述 UI 的轮廓和效果，此时事件驱动或 API 封装可以忽略。很多初学者在开发 UI 前往往忙于搜集各种小图片。漂亮的图标的确可以美化 UI，不过一般设计方式是编写易于扩展的 CSS。前期的 UI 呈现应尽量少使用图片，多用线条完成。本案例的设计草图如图 12.4 所示。

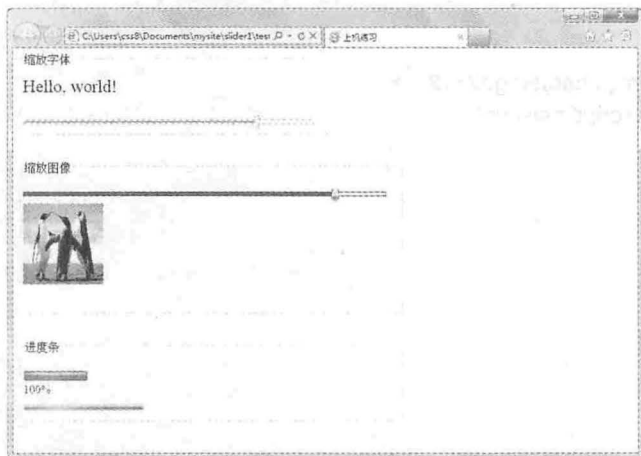


图 12.3 UI 滑动条演示效果

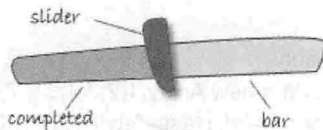


图 12.4 UI 设计草图

- ☑ slider: 表示作为拖曳手柄部件，用户可以通过拖曳此部分来更新 completed bar 的位置。
- ☑ completed: 表示作为 bar 的内嵌元素，作为特殊效果来显示 slider 与起始点的距离，亦即与 slider 的 value 值关联。
- ☑ bar: 表示 slider 的载体，completed 的满值。

设计思路：slider 作为手柄提供拖曳功能，作用区域为 bar，拖曳过程中 completed 区域必须实时更新，即动态改变长度，影响区域为 slider 至 bar 左端的距离。

3. 设计结构

开发 jQuery UI 在很多时候都需要与 UI 交互，因此在呈现上需要提供 HTML 文档树来绘制插件，最终通过 JavaScript DOM 来输出，那么在绘制简单的 DOM 结构时，会直接用 JavaScript 来完成。如果嵌套结构比较复杂，则应该先用 HTML 来完成，然后转变成 JavaScript 输出。

HTML 文档树如下:

```
<div class="defaultbar">
  <div class="jquery-completed"> </div>
  <div class="jquery-jslider"> </div>
</div>
```

- ☒ `<div class="defaultbar">` 标签表示 bar 部件。
- ☒ `<div class="jquery-completed">` 标签表示 completed 部件。
- ☒ `<div class="jquery-jslider">` 标签表示 slider 部件。

在前期 UI 呈现上最好不使用图片, 尽量使用线条、颜色来完成, 这样可以直接使用 CSS 代码进行控制。

代码如下:

```
/*---default skin---*/
.defaultbar{
  margin-top: 10px;
  height: 5px;
  background-color: #FFFFE0;
  border: 1px solid #A9C9E2;
  position: relative;
}
.defaultbar .jquery-completed{
  height: 3px;
  background-color: #7d9edb;
  top: 1px;
  left: 1px;
  position: absolute;
}
.defaultbar .jquery-jslider{
  height: 15px;
  background-color: #E6E6FA;
  border: 1px solid #A5B6C8;
  top: -6px;
  display: block;
  cursor: pointer;
  position: absolute;
}
```

将 bar 的 position 属性设置成 relative, 以方便子节点的浮动, 子节点使用 position: absolute 来获得内联浮动效果。此时使用 CSS 设计的 UI 效果如图 12.5 所示, 具备所需的元素 slider、completed 和 bar。

4. 设计规范

在正式编写 jQuery UI 代码之前, 应该进一步熟悉 jQuery 插件开发的一些规范 (第 9 章中曾经详细讲解过 jQuery 插件开发的方法和步骤)。

- ☒ 使用闭包

```
(function($) {
  //代码放在这儿
})(jQuery);
```

这是来自 jQuery 官方的插件开发规范要求, 使用这种方式编写的好处如下:

- ☒ 避免全局依赖

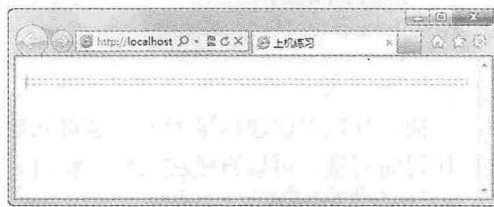


图 12.5 初步设计的 UI 底图

☑ 避免第三方破坏

☑ 兼容 jQuery 操作符\$和 jQuery

这段代码在被解析时与下面代码相同:

```
var jq = function($) {
    //代码放在这儿
```

```
};
```

```
jq(jQuery);
```

☑ 扩展

jQuery 提供了两个供用户扩展的基类: \$.extend()和\$.fn.extend()。\$.extend()用于扩展自身方法,如\$.ajax、\$.getJSON等;\$.fn.extend()则是用于扩展 jQuery 类,包括方法和对 jQuery 对象的操作。为了保持 jQuery 的完整性,建议多使用\$.fn.extend 进行插件开发,而尽量少使用\$.extend()。

☑ 选择器

jQuery 提供了功能强大并兼容多种 CSS 版本的选择器,不过在使用选择器时要注重效率问题。

尽量使用 ID 选择器, jQuery 的选择器使用的 API 都是基于 getElementById()或 getElementsByTagName()原生方法的。ID 选择器的执行效率是最高的,因为 jQuery 会直接调用 getElementById()方法去获取 DOM 元素,而通过样式选择器获取 jQuery 对象时,往往会使用 getElementsByTagName()方法获取,然后进行筛选。

样式选择器应该尽量明确指定标签名称。如果开发人员使用样式选择器来获取 DOM 元素,且这些元素属于同一类型,如获取所有 className 为 jquery 的 div,那么应该使用的写法是\$('div.jquery'),而不是\$('.jquery')。这样写的好处非常明显,在获取 DOM 元素时, jQuery 会获取 div 元素,然后进行筛选,而不是获取所有 DOM 元素,再执行筛选。

很多初学者在使用 jQuery 获取指定上下文中的 DOM 元素时,喜欢使用迭代方式,如\$('.jquery.child'),获取 className 为 jquery 的 DOM 下的所有 className 为 child 的节点。其实这样编写代码付出的代价是非常大的, jQuery 会不断地进行深层遍历来获取需要的元素,即使确实需要,应该使用诸如\$(selector,context)、\$(selector1>selector2)、\$(selector1).children(selector2)、\$(selector1).find(selector2)之类的方式。

5. 编写代码

对 UI 有了清晰的认识后,就可以使用 JavaScript 输出 HTML 了。这里使用 jSlider 来命名这个 slider UI。为了避免插件冲突,插件命名时应谨慎。代码如下:

```
$.extend($.fn, {
    ///<summary>
    ///apply a slider UI
    ///</summary>
    jSlider: function(setting) {
    }
});
```

在插件开发中比较标准的方式是将元数据独立出来,并开放 API,如 setting 参数传入值。有时候为了减少代码编写量,可以直接在插件内赋值。代码如下:

```
var ps = $.extend({
    renderTo: $(document.body),
    enable: true,
    initPosition: 'max',
    size: { barWidth: 200, sliderWidth: 5 },
    barCssName: 'defaultbar',
    completedCssName: 'jquery-completed',
    sliderCssName: 'jquery-jslider',
    sliderHover: 'jquery-jslider-hover',
    onChanging: function() {}},
```

```
onChanged: function() { }
}, setting);
```

规范的做法如下：

```
$.fn.jSlider.default = {
  renderTo: $(document.body),
  enable: true,
  initPosition: 'max',
  size: { barWidth: 200, sliderWidth: 5 },
  barCssName: 'defaultbar',
  completedCssName: 'jquery-completed',
  sliderCssName: 'jquery-jslider',
  sliderHover: 'jquery-jslider-hover',
  onChanging: function() { },
  onChanged: function() { }
};
```

```
$.extend({}, $.fn.jSlider.default, setting);
```

API 描述如下。

- ☑ renderTo: jSlider 的载体、容器，可以是一个 jQuery 对象，也可以是选择器。
- ☑ enable: jSlider 插件是否可用，true 时 end-user 可拖曳，否则禁止。
- ☑ initPosition: jSlider 的初始值，max 或者 min，亦即 slider 的 value 值，1 或者 0。
- ☑ size: jSlider 的参数，包括两个值，barWidth - bar 的长度和 sliderWidth - slider 的长度。
- ☑ barCssName: bar 的样式名称，便于 end-user 自行扩展样式。
- ☑ completedCssName: completed 的样式名称。
- ☑ sliderCssName: slider 的样式名称。
- ☑ sliderHover: slider 聚焦时的样式名称。
- ☑ onChanging: slider 被拖曳时触发的事件。
- ☑ onChanged: slider 拖曳结束时触发的事件。

将 renderTo 强制转换成 jQuery 对象：

```
ps.renderTo = (typeof ps.renderTo == 'string' ? $(ps.renderTo) : ps.renderTo);
```

然后将 HTML 结构树输出到 render 中：

```
var sliderbar = $('<div><div>&nbsp;</div><div>&nbsp;</div></div>')
    .attr('class', ps.barCssName)
    .css('width', ps.size.barWidth)
    .appendTo(ps.renderTo);
var completedbar = sliderbar.find('div:eq(0)')
    .attr('class', ps.completedCssName);
var slider = sliderbar.find('div:eq(1)')
    .attr('class', ps.sliderCssName)
    .css('width', ps.size.sliderWidth);
```

HTML 结构树说明如下：

```
<div>                                <!-- sliderbar -->
  <div>&nbsp;</div>                        <!-- completed bar -->
  <div>&nbsp;</div>                        <!-- slider -->
</div>
```

这样就在 UI 上直接呈现了 HTML 结构，并且用定制的 CSS 进行渲染，分别用 sliderbar、completedbar 和 slider 对需要的 3 个对象进行缓存。

在呈现了 UI 之后，就需要提供方法来实现 slider 的拖曳功能。在这之前还需要实现一个方法，就是 completedbar 的实时更新，即在拖动 slider 的时候让 completedbar 始终填充左侧区域。代码如下：


```

var bw = sliderbar.width(), sw = slider.width();
ps.limited = { min: 0, max: bw - sw };
if (typeof window.$sliderProcess == 'undefined') {
    window.$sliderProcess = new Function('obj1', 'obj2', 'left',
        'obj1.css('left',left);obj2.css('width',left);');
}
$sliderProcess(slider, completedbar, eval('ps.limited.' + ps.initPosition));

```

bw 和 sw 用来存储 sliderbar 和 slider 的长度，此处没有直接使用 ps.size 里的值，是为了防止样式里的 border-width 对 width 造成的破坏。

定义一个私有成员 limited 来存储 slider[left] 的最大值和最小值，并在后面直接使用 eval('ps.limited.' + ps.initPosition) 来获取，从而避免 switch 操作。同时还需定义一个全局 Function 用来定位 completedbar 的填充长度以及 slider 左侧距离，命名为 \$sliderProcess。

接下来的工作就是 slider 的拖曳功能了。详细代码如下：

```

//drag and drop
var slide = {
    drag: function(e) {
        var d = e.data;
        var l = Math.min(Math.max(e.pageX - d.pageX + d.left, ps.limited.min), ps.limited.max);
        $sliderProcess(slider, completedbar, l);
        //push two parameters: 1st:percentage, 2nd: event
        ps.onChanging(l / ps.limited.max, e);
    },
    drop: function(e) {
        slider.removeClass(ps.sliderHover);
        //push two parameters: 1st:percentage, 2nd: event
        ps.onChanged(parseInt(slider.css('left')) / ps.limited.max, e);
        $().unbind('mousemove', slide.drag).unbind('mouseup', slide.drop);
    }
};
if (ps.enable) {
    //bind events
    slider.bind('mousedown', function(e) {
        var d = {
            left: parseInt(slider.css('left')),
            pageX: e.pageX
        };
        $(this).addClass(ps.sliderHover);
        $().bind('mousemove', d, slide.drag).bind('mouseup', d, slide.drop);
    });
}

```

这样当 jSlider enable 属性为 true 时，在 end-user 按下鼠标时绑定 mousemove 事件，在鼠标弹起时移除，只需要同步更新 slider 的 left 属性和 completedbar 的 width 即可，同时在 drag 中绑定 onChanging 方法，在 drop 中绑定 onChanged 方法，向这两个方法推送的参数相同——百分比，即 value 值，取值范围为 0~1。那么 jSlider UI 就基本成型了，即可完成一个可拖曳的滑动条。

6. 扩展

这里为用户公开一个方法，用于设置 jSlider 的 value，首先考虑的是作为方法需要一个作用对象(jSlider)，这里不想将作用对象作为参数传入，而且将这个方法作为插件来开发，命名为 setSliderValue，开放两个参数：value 值和 callback（设置完成后的回调函数）。代码如下：

```
$.fn.setSliderValue(v,callback);
```

由之前的设计可知, 在 slider 拖动时主要作用于两个对象——slider 和 completedbar, 那么在 jSlider 插件末尾加上一段代码来返回 slider 对象:

```
slider.data = { bar: sliderbar, completed: completedbar };
return slider;
```

这样在初始化 jSlider 时就可以直接用一个变量来获取 jSlider 对象, 然后调用 setSliderValue 方法。代码如下:

```
var slider = $.fn.jSlider({});
slider.setSliderValue(v,function({}));
setSliderValue 代码:
try {
    //validate
    if (typeof v == 'undefined' || v < 0 || v > 1) {
        throw new Error("'v' must be a Float variable between 0 and 1.");
    }
    var s = this;
    //validate
    if (typeof s == 'undefined' ||
        typeof s.data == 'undefined' ||
        typeof s.data.bar == 'undefined') {
        throw new Error("You bound the method to an object that is not a slider!");
    }
    $sliderProcess(s, s.data.completed, v * s.data.bar.width());
    if (typeof callback != 'undefined') { callback(v); }
}
catch (e) {
    alert(e.message);
}
```

这里调用了全局函数 \$sliderProcess(), 在设置 slider 的 value 值时, 进行 completedbar[width]和 slider[left]的更新。由于此处进行了异常处理, 所以如果 end-user 在确保 setSliderValue 被作用于 jSlider 对象时可以删除此异常处理代码。

7. 皮肤

根据 jSlider 的 API 规范, 可以更加方便地为其设定皮肤。为了让 jSlider 更加专业, 这里需要两张图片, 如图 12.6 所示。

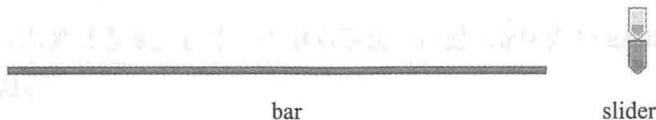


图 12.6 素材图

其中, bar.gif 用来作为 completedbar 背景, slider.gif 用来作为 slider 背景, 然后重新更新样式。代码如下:

```
/*----blue skin----*/
.bluebar{
    margin-top: 10px;
    height: 4px;
    background:#F7F7F7;
    border:solid 1px #3e3e3e;
    position: relative;
```

```

}
.bluebar .jquery-completed{
    height: 4px;
    background:url(images/slider/blue/bar.gif) left center no-repeat;
    top: 0;
    left:0;
    position: absolute;
}
.bluebar .jquery-jslider{
    height: 17px;
    background:url(images/slider/blue/slider.gif) center 0 no-repeat;
    top: -4px;
    display: block;
    cursor: pointer;
    position: absolute;
}
}
.bluebar .jquery-jslider-hover{
    background-position:center -17px;
}
}

```

由于在设置样式时，设置子节点样式使用了 API 的默认值，因此在创建 jSlider 时，只需要设置 barCssName 即可。代码如下：

```

var blue = $.fn.jSlider({
    renderTo: 'slidercontainer',
    size: { barWidth: 500, sliderWidth: 10 },
    barCssName: 'bluebar',
    onChange: function(percentage, e) {
        //code goes here
    }
});

```

呈现出来的 UI 效果如图 12.7 所示。

设置值如下：

```

//set percentage with a callback function
blue.setSliderValue(0.65, function(percentage) {
    //code goes here
});

```

当然，不仅可以将 jSlider 作为 slider 使用，也可以作为一个进度条进行使用。演示效果如图 12.8 所示。

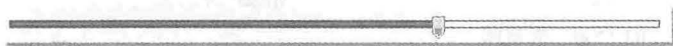


图 12.7 设计的 UI 效果图



图 12.8 UI 进度条效果图

12.2 使用 jQuery UI 库

jQuery 官网提供了众多插件，访问 <http://plugins.jquery.com/> 可以下载这些插件，如图 12.9 所示，其中的插件数量还在不断增加。当然，用户也可以把个人开发的插件上传到 jQuery 官网，同时对喜爱的插件进行评级和评价。

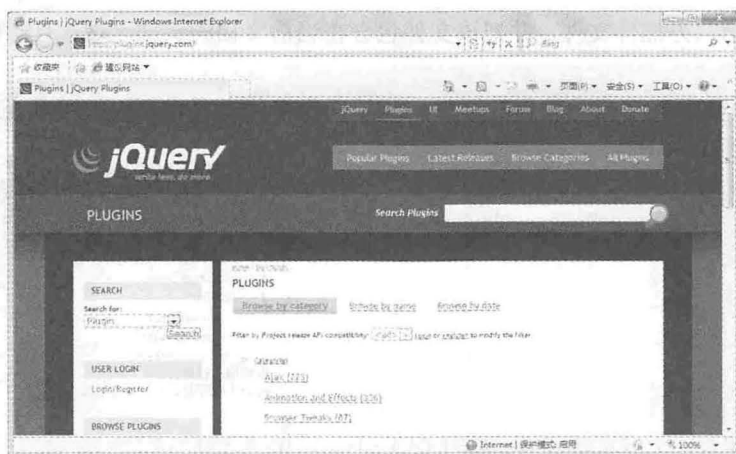


图 12.9 jQuery 官网插件

12.2.1 认识 jQuery 插件库

jQuery 对所有插件进行了分类,通过这些分类读者可以了解插件开发和应用的主体范围,并能够根据实际开发需要有选择地下载这些插件。注意,分类列表中的数字表示该类插件的数目,该数目是一个动态数字,每天都会发生变化。

- ☑ Ajax (223): 异步通信插件。
- ☑ Animation and Effects (336): 动画和特效插件。
- ☑ Browser Tweaks (87): 浏览器系统调整插件。
- ☑ Data (163): 数据处理插件。
- ☑ DOM (165): 文档对象模型相关插件。
- ☑ Drag-and-Drop (38): 拖放插件。
- ☑ Events (143): 事件插件。
- ☑ Forms (397): 表单插件。
- ☑ Integration (118): 整合插件,即功能综合性质的插件。
- ☑ JavaScript (162): JavaScript 核心功能插件。
- ☑ jQuery Extensions (249): jQuery 扩展功能插件。
- ☑ Layout (204): 布局插件。
- ☑ Media (145): 媒体插件。
- ☑ Menus (114): 菜单插件。
- ☑ Metaplugin (27): 元插件,即所谓的插件的插件,即用来制作插件的插件。
- ☑ Navigation (170): 导航器插件。
- ☑ Tables (82): 数据表格插件。
- ☑ User Interface (698): 用户界面插件,或者说是窗口或面板插件。
- ☑ Utilities (366): 公用工具插件。
- ☑ Widgets (263): 小部件插件,小的应用程序。
- ☑ Windows and Overlays (115): 窗口和遮盖插件。

12.2.2 使用外部插件

使用外部插件比较简单,具体操作步骤如下。

(1) 在网上下载第三方插件文件, 并把该文件包含在当前文档的头部区域, 且确保位于 jQuery 核心源文件的后面。

例如, 下面是一个数据表格拖动效果的插件。当按住表格行之后, 可以拖动其在表格中的位置。演示效果如图 12.10 和图 12.11 所示。

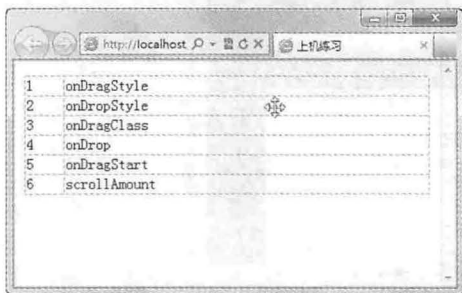


图 12.10 拖动表格行



图 12.11 拖动后的表格行效果

本插件在 jQuery 官网上的下载地址为 <http://plugins.jquery.com/project/TableDnD>。

本插件作者的官方网址为 <http://www.isocra.com/2008/02/table-drag-and-drop-jquery-plugin/>。

(2) 下载并解压该插件, 然后在当前页面中导入该文件。代码如下:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jQuery/jquery.tablednd_0_5.js" type="text/javascript"></script>
</head>
```

(3) 在当前文档中使用该插件创建应用实例, 或者调用该插件的扩展方法。一般可以放置在页面初始化事件处理函数中。在应用该插件之前, 应该在文档中插入一个数据表格, 并把数据表格的 id 传递给 jQuery 选择器, 然后在选择器上面调用 tableDnD() 插件函数。

每一种插件根据功能大小, 都会包含一些参数说明, 或者使用帮助文档, 读者可以在插件的脚本文件顶部的注释说明中进行查阅, 也可以在插件的官方网址上进行查询。例如, 本表格拖曳插件可以参阅 <http://www.isocra.com/2008/02/table-drag-and-drop-jquery-plugin/> 网址了解其参数及其应用案例。

(4) 在文档中插入一个表格结构。代码如下:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jQuery/jquery.tablednd_0_5.js" type="text/javascript"></script>
<style type="text/css">
table { border:solid 1px #6CF; width:100%; border-collapse:collapse; }
td { border:solid 1px #6CF; }
.drag { background-color:#6CF; }
</style>
<title>上机练习</title>
</head>
```

```

<body>
<table id="table1">
  <tr id="1">
    <td>1</td>
    <td>onDragStyle</td>
  </tr>
  <tr id="2">
    <td>2</td>
    <td>onDropStyle</td>
  </tr>
  <tr id="3">
    <td>3</td>
    <td>onDragClass</td>
  </tr>
  <tr id="4">
    <td>4</td>
    <td>onDrop</td>
  </tr>
  <tr id="5">
    <td>5</td>
    <td>onDragStart</td>
  </tr>
  <tr id="6">
    <td>6</td>
    <td>scrollAmount</td>
  </tr>
</table>
<div id="debugArea"></div>
</body>
</html>

```

(5) 设置 tableDnD() 插件函数包含 3 个插件选项参数。代码如下:

```

<script type="text/javascript">
$(function(){
    //页面初始化处理函数
    $("#table1").tableDnD({
        //调用表格拖动函数
        onDragClass: "drag", //设置拖动表格行的类样式
        onDrop: function(table, row) {
            //设置放下拖动表格行后触发的回调函数，默认包含两个参数，第 1 个
            //参数传递当前表格对象，第 2 个参数传递当前行对象

            var rows = table.tBodies[0].rows;
            var debugStr = "当前拖动行是: "+row.id+"<br />拖动后顺序是: ";
            for (var i=0; i<rows.length; i++) {
                debugStr += rows[i].id+" ";
            }
            $("#debugArea").html(debugStr);
        },
        onDragStart: function(table, row) {
            //设置拖动表格行后触发的回调函数，默认包含两个参数，第 1 个
            //参数传递当前表格对象，第 2 个参数传递当前行对象
            $("#debugArea").html("当前拖动行是: "+row.id);
        }
    });
});

```

```
})
</script>
```

很多插件都具有很强的灵活性,并提供一些供用户设置的可选参数,用来改变插件的行为和属性,这样就可以按照特殊需要自定义插件使用,也可以简单保持其默认的行为。例如,在上面表格拖曳插件中,可以直接调用 `tableDnD()` 函数。代码如下:

```
$("#table1").tableDnD() //调用表格拖动函数
```

12.2.3 认识 UI 插件

jQuery UI 源自一个 Interface 插件,它是由 Stefan Petre 创作的。最初的 Interface 插件只支持 jQuery 1.1.2 版本,版本号为 1.2。后来在 Paul Bakaus 的参与和领导下,对 Interface 插件的源代码进行重构,并统一和规范了 API 接口和文档,更名为 jQuery UI 1.5。从此, jQuery UI 确定了官方插件的地位,官方访问地址为 <http://jqueryui.com/>。

jQuery UI 是未来 jQuery 技术框架发展的趋势,也是未来互联网客户端发展的方向。jQuery UI 包含以下 3 部分。

☑ 交互

包括与鼠标的交互,与键盘的交互,如拖放、缩放、选择和排序等基本交互行为。Web 部件中很多行为都是基于这些基本交互来设计的。交互操作需要导入 jQuery UI 核心库 `ui.core.js` 文件。

☑ 部件

部件包含各种界面风格和形式,如导航、对话框、提示、面板、侧栏、滑块、树结构、日历、拾色器、放大镜、标签、自动完成、进度条、微调控制器、历史、布局、栅格、菜单、工具提示、工具栏、上传组件等。部件需要导入 jQuery UI 核心库 `ui.core.js` 文件。

☑ 效果

效果包含各种动画效果,需要导入效果库文件 `effects.core.js`。

如果读者觉得这个 UI 插件包比较麻烦,可以直接单击右上角的 Build custom download 超链接(如图 12.12 所示),打开自定义组件下载(<http://jqueryui.com/download>),如图 12.13 所示。在左侧选择需要的组件,包括 UI Core (UI 核心库)、Interactions (交互)、Widgets (部件)、Effects (效果)。然后,在右侧 Theme (主题)中选择一种组件的样式主题,在右侧下面的 Version (版本)中选择框架的版本。最后单击 Download 按钮,即可按需下载对应的组件。

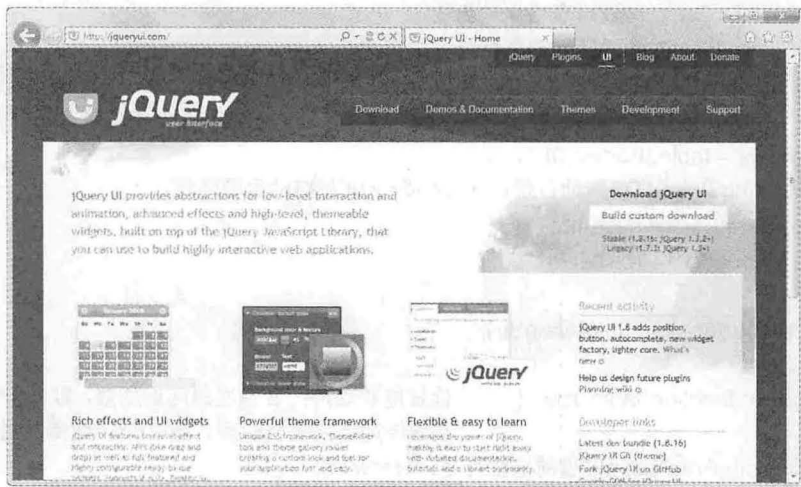


图 12.12 jQuery UI 插件官方下载页面



图 12.13 自定义下载 jQuery UI 插件

12.2.4 建立开发环境

jQuery UI 插件下载地址为 <http://jqueryui.com/>，单击页面右侧的稳定版本 1.8.16（即 Stable (1.8.16: jQuery 1.3.2+)），最新版本是 1.8.16，如图 12.14 所示。下载完毕，解压 jquery-ui-1.8.16.custom.rar 文件，就可以在 jquery-ui-1.8.16.custom 文件中看到欢迎页面 index.html 和 3 个子目录（js、CSS 和 development-bundle），如图 12.14 所示。为了方便引用，可以把 jquery-ui-1.8.16.custom 更名为 jqueryui 文件夹。

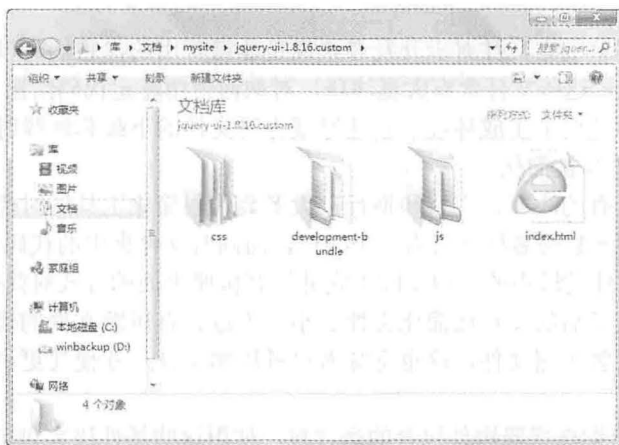


图 12.14 解压目录

12.2.5 jQuery UI 库结构

在解压的库压缩文件中，可以看到 development-bundle 文件夹，可以看到 UI 插件的示例（demos）、文档（docs）、主题（themes）和核心库文件（ui）和扩展（external）文件夹。该文件夹包含内容如下：

- ☒ demos 目录
- ☒ docs 目录
- ☒ external 目录
- ☒ themes 目录

- ☑ ui 目录
- ☑ AUTHORS.txt 文件
- ☑ GPL-LICENSE.txt 文件
- ☑ jquery-1.6.2.js 文件
- ☑ MIT-LICENSE.txt 文件
- ☑ version.txt 文件

demos 目录展示了一系列功能范例，以及在实现应用中如何使用各 UI 组件的例子。每个组件都有各自的演示界面，这些页面都被设计为可以本地访问。

功能范例页面说明了每个组件的基本实现，并展示了一些常见的配置，这些页面与在线的范例网页是一样的。

现实应用中的例子强调了组件的特定功能，并在页面中展示此功能，但是很少或者根本就没有说明文字。虽然这些例子与 jQuery UI 项目主页的范例相同，但是展示的效果不如在线的范例页面。

themes 目录中包含了组件主题。开发库提供了两种主题：base 和 smoothness。base 主题以单色为主，而 smoothness 提供了过渡、平滑等多色效果。这两个主题为每个高层控件都提供样式，并且如果需要，可以完全不加修改地独立使用。

这些主题中的一些 CSS 文件不仅定义了控件的外观，还与控件的功能相关。因此，如果需要为特定控件提供定制的皮肤，可以有两种选择。一是忽略控件原有的皮肤文件，完全使用自己的 CSS 文件来代替相应主题文件，二是简单修改特定的用于处理外观的规则。

第 1 种方式虽然可行，但是需要用户花费大量的时间来设计独立的皮肤，而第 2 种是建立在现有主题样式的基础上，适当修改，可以提高代码利用率，节省时间。

ui 目录中包含了每种组件和特效的所有未简化版代码文件，其中几个子目录包含了简化和打包后的组件，以及 i18n 文件夹。

每种库组件和特效的完整版本文件对于开发者来说是非常有用的，它们可以被打开并阅读，以便更好地了解特定组件的工作方式，这些文件含有大量注释，对如何使用特定代码给出了建议。

而每种组件的简化版本适用于生成环境，它能够减少对文件的下载和解释时间，借助于快速开发的各种工具，JavaScript 可以很容易被简化。

简化后的文件删除了所有的注释、空格和换行，大多数代码简化工具还对代码进行了混淆处理，如有可能，可以将对象、变量和函数的名称缩写为一个字母，而同时文件夹中的代码仍保留原来的功能。

每种组件的打包版本文件是最小的，但实际上它并没有按照上面的方式对简化文件进行进一步的简化，而只是对其进行压缩，使压缩后的文件比简化文件更小。不过含有压缩文件的代码需要进行一些改变，即需要添加一些客户端代码来解压缩文件，这也意味着尽量压缩文件尺寸使其更小，但通常需要更长时间进行解释。

i18n 目录中放置的是日期选择器控件包含的语言包，使用这些插件语言包可以轻松地对日期选择器进行国际化。

12.2.6 主题定制器

主题定制器是用 jQuery 编写的主题定制工具，用于可视化地制作自定义的 jQuery UI 主题，然后将其打包成一个易于使用的可直接下载的文件，并且可以直接在项目中使用而无须额外的编码。

主题定制器是由 Filament Group Inc 创建的，它使用了大量发布于开源社区的 jQuery 插件，详细信息请参阅 <http://jqueryui.com/themeroller/>。

使用主题定制器可以快速而简便地创建完整的主题，其中包含所有目标元素所需要的样式，以及所有应用所需要的图片，并且该主题与所有 jQuery 稳定版兼容。如图 12.15 所示，显示了主题定制器的接口，

它是非常容易使用的。

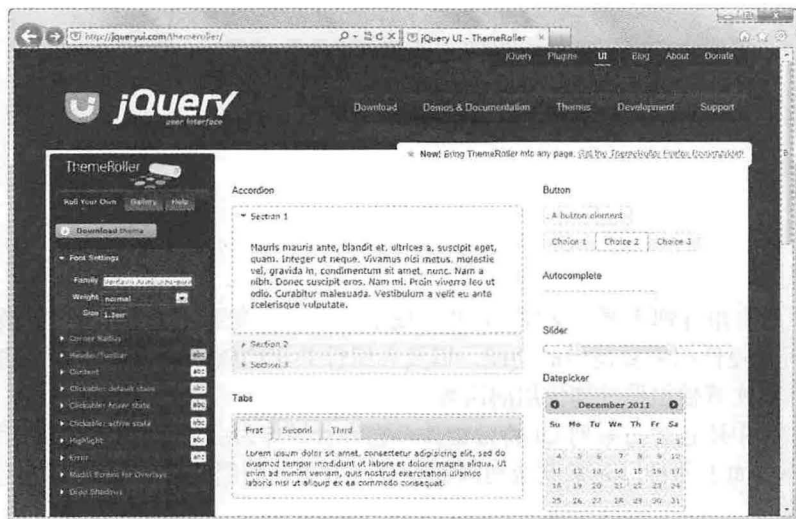


图 12.15 主题定制器接口

考虑到用户可能没有特别兴趣创建主题，该页面还提供了一系列预先配置好的主题，它们可以被立即使用。除了方便使用之外，预先选择主题所具有的最大好处就是，一旦用户选择了一个主题，该主题将会在主题定制器的首页自动被加载，然后用户可以很轻松地根据需要稍加修改后使用。

12.2.7 如何使用 jQuery UI 组件

jQuery 从 1.5 版本开始，简化了各种组件的 API，使程序库更容易使用，并且具有更加强大的功能。如果有过在程序库中使用某种组件的经验，那么在使用其他组件时也会非常容易上手，因为每个组件的方法几乎都是以同样的方式被调用的。

所有的组件都以一致的方式调用方法，即将方法名作为字符串传递给组件构造函数，同时方法需要的各种参数也同样以字符串的方式随后传递。

例如，在选项卡组件中调用 `destroy()` 方法，可以使用下面的简易方式：

```
$("#id").tabs("destroy");
```

所有其他组件所公开的单个方法也同样是用这种简单的方式调用的。使用 jQuery UI 的感觉如同使用 jQuery 本身一样，因此对于已经使用 jQuery 编写过可靠代码的程序员来说，下一步转向 jQuery UI 是合理的选择。

许多组件共享一个所需开放的函数方式集合，如库中每个单个组件都具有 `destroy`、`enable` 和 `disable` 方法，以及许多其他类似的公开函数，这同样使每个组件都能够以非常简单、直观的方式被使用。

12.2.8 组件类别

jQuery UI 库中有两种组件，底层的交互助手是用来处理鼠标标准事件，并在页面上产生可见的对象的控件，每种交互助手都是为执行某种特定功能而设计的。

交互助手的分门别类，实际上形成了程序库的基本核心部分，其中包含下列组件：

- ☒ 拖动 (draggable)。
- ☒ 放置 (droppable)。
- ☒ 变化尺寸 (resizable)。

- ☒ 选择 (selectable)。
- ☒ 排序 (sortable)。

下面的高层控件通常建立在底层控件所提供的基础之上：

- ☒ accordion。
- ☒ 自动完成。
- ☒ 日期选择器。
- ☒ 对话框。
- ☒ 滑动条。
- ☒ 选项卡。

ui.core.js 是其他所有组件都需要的文件，它并不属于任何一种类别，但是仍然可以被视为一种特殊的组件。该文件构建了所有控件都需要使用的功能，以及各组件共享的核心函数。但是这些函数不能够直接使用，所以没有开放任何在其他组件外部使用的函数。

除了这些组件，库中还包含一系列 UI 特效组件，它们曾经作为名为 Enchant 的助手库而完全独立地存在。这些特效组件为页面上的目标元素制造动画或者过渡效果，利用它们为页面上功能固定的组件添加风格和样式是非常棒的。


12.2.9 浏览器支持

与 jQuery 一样，jQuery UI 支持当前主流的浏览器，包括 IE 6+、Firefox 2+、Opera 9+、Safari 3+ 和 Chrome。jQuery UI 几乎支持任何普通的 Web 浏览器，库中的组件和控件都由语义正确的 HTML 生成，因此页面中不会创建或者使用多余的、不必要的元素。

读者在编写代码时，应该模仿 jQuery UI 风格，努力维护一个开放的内容内核，如表现层和功能层。

第13章

jQuery UI 交互开发

( 视频讲解：40 分钟)

交互开发是一组内在的行为，以满足常见的应用需求。尽管不能够直接在页面上看到这些组件，但是它们给不同元素带来的效果及行为是显而易见的。与高层的控件不同，它们属于底层组件。jQuery UI 交互组件包括 5 种，每种都适用于特定的交互场合。说明如图 13.1 所示。

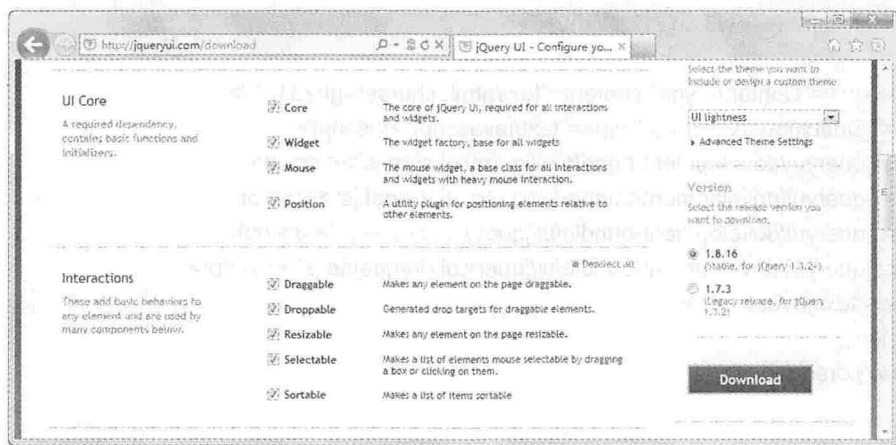


图 13.1 jQuery UI Interactions

对于用户来说，交互组件使得页面中所适用的元素具有更富吸引力的外观和更好的交互性，这会为站点添加价值或者帮助 Web 应用显得更为专业。此外对于应用平台来说，它们还模糊了浏览器和桌面之间的界限，使得浏览器应用也能具有与桌面应用同样丰富的界面与交互手段。

13.1 拖 放

拖放实际上是两个动作：拖动 (Draggable) 和投放 (Droppable)。拖动组件能够将任何特定元素转化为访问者可以通过鼠标指针在页面上拖动的组件。该组件所开放的方式提供了限制拖动的移动方式，在松开鼠标时让元素回到开始位置，以及其他很多功能。投放组件能够在页面上定义一个区域或者一个容器，以便访问者将拖动的元素投放到其中，从而完成某种功能，如界定所选择的内容，在购物篮中添加选购的商品等。投放可以触发一系列事件，可以对拖动交互中所感兴趣的时间点进行响应。

拖动和投放是两个相互关联的组件，其中一个发生，另一个总是关闭的，在使用时应该结合这两个组件。虽然可以仅设计拖动动作，如对话框的拖动等，但是如果设置投放目标，拖动是没有任何效果的。这两个组件只被设计用于简单的拖动和投放场景中，它们本身就是比较复杂的 Web 交互过程。

13.1.1 拖动对象

拖动组件能够使任何特定元素或者元素集合变为可拖动，即访问者可以选择它们并在页面上移动。拖动是一种非常实用的效果，它能够用于很多场合以提升 Web 页面的交互性，使用 jQuery UI 库可以不需要担心浏览器之间的细微差异，这些细微差异对于实现和维护 Web 页面上的拖动元素来说是巨大的障碍。

实现拖动行为需要引入下面几个 JavaScript 文件，然后为拖动目标元素绑定 `draggable()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.mouse.js
- ☒ jquery.ui.draggable.js

【示例 1】 为插入图片的包含框绑定拖动行为，实现让鼠标自由拖动图标。演示效果如图 13.2 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<script type="text/javascript">
$(function(){
    $("#box").draggable();
})
</script>
<style type="text/css">
#box img { width:150px; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="box"></div>
</body>
</html>
```

如果仅激活拖动功能，只需要引入 UI 库中的下列 3 个文件：

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.draggable.js

拖动组件具有大量可以配置的属性，使用这些属性，用户能够精确控制拖动行为，说明如表 13.1 所示。

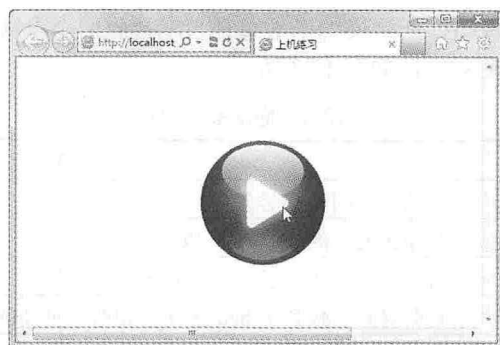


图 13.2 基本拖动效果

表 13.1 拖动组件的属性

属 性	说 明
appendTo	为可拖动的元素指定容器元素
axis	限制拖动只能够沿着某个方向进行，可接受字符串 x 和 y 作为属性值
cancel	阻止特定的元素被拖动，如果它们匹配某个选择条件
containment	阻止可拖动元素被拖动到它的父元素边界之外
cursor	指定拖动光标的 CSS 样式
cursorAt	在元素被拖动时，指定显示拖动状态的光标的默认相对位置
delay	指定开始拖动时延迟的毫秒数
distance	指定在可拖动元素中按下鼠标按钮后指针所要求移动的距离（像素值），即移动大于此距离后才开始拖动该元素
grid	使可拖动元素只能够在页面中的虚拟网格中移动
handle	确定可拖动元素中用于防止拖动指针的具体区域
helper	定义一个虚构的拖动元素，在拖动时替代实际的可拖动元素
opacity	设置 helper 元素的透明度
rever	使可拖动元素在拖动结束时返回开始位置
scroll	使可拖动元素可以自动卷动
scrollSensitivity	定义可拖动元素在距离容器边界多近时才开始卷动容器
scrollSpeed	拖动元素时容器的卷动速度
snap	使拖动对象在到达特定元素边缘时自动与之靠拢
snapMode	可以被设置为 inside、outside、both，默认为 both
snapTolerance	可拖动对象离目标对象的距离低于此像素值时开始靠拢
refreshPositions	在每次鼠标移动时计算可拖动的位置
zindex	设置 helper 元素的 z 索引

除了上面介绍的属性外，拖动组件还提供了 3 个事件，专门用于特定事件执行代码的回调函数，说明如表 13.2 所示。

表 13.2 拖动组件的事件

事 件	说 明
drag	拖动过程中鼠标移动
start	拖动开始时
stop	拖动结束时

当定义回调函数使用这些事件时，所定义的函数总是自动接收两个参数，第 1 个参数为原始的事件对象，第 2 个参数为包含下列属性的对象，如表 13.3 所示。

表 13.3 拖动参数对象

参 数 项	说 明
options	用于拖动组件的配置对象
helper	代表 helper 元素的 jQuery 对象
position	一个嵌套对象，包含了 helper 元素相对于原始可拖动对象的 top 和 left 属性值
absolutePosition	一个嵌套对象，包含了 helper 元素相对于页面的 top 和 left 属性值

拖动组件定义 3 个基本方法，不包括其构造方法，如表 13.4 所示。

表 13.4 拖动组件的方法

方 法	说 明
destroy()	彻底清除该控件
enable()	根据索引号激活被禁用的拖动功能
disable()	根据索引号禁用拖动功能

1. 拖动事件侦查

下面示例演示了如何调用拖动组件。演示效果如图 13.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    var $start_counter = $( "#event-start" ),
        $drag_counter = $( "#event-drag" ),
        $stop_counter = $( "#event-stop" ),
        counts = [ 0, 0, 0 ];
    $( "#draggable" ).draggable({
        start: function() {
            counts[ 0 ]++;
            updateCounterStatus( $start_counter, counts[ 0 ] );
        },
        drag: function() {
            counts[ 1 ]++;
            updateCounterStatus( $drag_counter, counts[ 1 ] );
        },
        stop: function() {
            counts[ 2 ]++;
            updateCounterStatus( $stop_counter, counts[ 2 ] );
        }
    });
});
```

```

    }
  });
  function updateCounterStatus( $event_counter, new_count ) {
    if ( !$event_counter.hasClass( "ui-state-hover" ) ) {
      $event_counter.addClass( "ui-state-hover" )
        .siblings().removeClass( "ui-state-hover" );
    }
    $( "span.count", $event_counter ).text( new_count );
  }
});
</script>
<style type="text/css">
#draggable { width: 16em; padding: 0 1em; }
#draggable ul li { margin: 1em 0; padding: 0.5em 0; }
* html #draggable ul li { height: 1%; }
#draggable ul li span.ui-icon { float: left; }
#draggable ul li span.count { font-weight: bold; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="draggable" class="ui-widget ui-widget-content">
  <p>拖动事件</p>
  <ul class="ui-helper-reset">
    <li id="event-start" class="ui-state-default ui-corner-all"><span class="ui-icon ui-icon-play"></span>
"start" 被调用 <span class="count">0</span>x</li>
    <li id="event-drag" class="ui-state-default ui-corner-all"><span class="ui-icon ui-icon-arrow-4"></span>
"drag" 被调用 <span class="count">0</span>x</li>
    <li id="event-stop" class="ui-state-default ui-corner-all"><span class="ui-icon ui-icon-stop"></span>
"stop" 被调用 <span class="count">0</span>x</li>
  </ul>
</div>
</body>
</html>

```

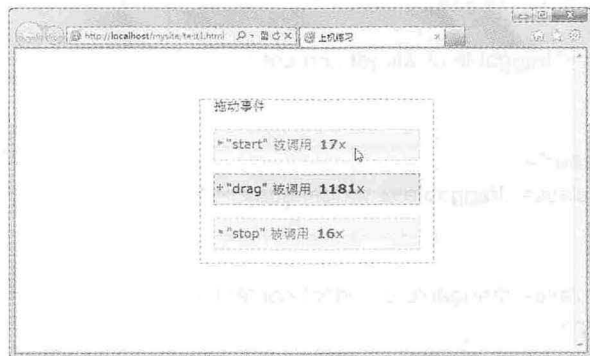


图 13.3 事件侦查

2. 设置移动范围

下面示例演示了如何为移动元素设置可移动范围，移动范围可以包括父元素或者包含框等，也可以设置可移动方向，如水平移动或者垂直移动。演示效果如图 13.4 所示。


```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("#draggable").draggable({ axis: "y" });
    $("#draggable2").draggable({ axis: "x" });
    $("#draggable3").draggable({ containment: "#containment-wrapper", scroll: false });
    $("#draggable4").draggable({ containment: "#demo-frame" });
    $("#draggable5").draggable({ containment: "parent" });
});
</script>
<style type="text/css">
.draggable { width: 90px; height: 90px; padding: 0.5em; float: left; margin: 0 10px 10px 0; }
#draggable, #draggable2 { margin-bottom: 20px; }
#draggable { cursor: n-resize; }
#draggable2 { cursor: e-resize; }
#containment-wrapper { width: 95%; height: 150px; border: 2px solid #ccc; padding: 10px; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="demo">
<div id="draggable" class="draggable ui-widget-content">
    <p>垂直移动</p>
</div>
<div id="draggable2" class="draggable ui-widget-content">
    <p>水平移动</p>
</div>
<div id="containment-wrapper">
    <div id="draggable3" class="draggable ui-widget-content">
        <p>框内移动</p>
    </div>
    <div id="draggable4" class="draggable ui-widget-content">
        <p>外框内移动</p>
    </div>
    <div class="draggable ui-widget-content">
        <p id="draggable5" class="ui-widget-header">内框内移动</p>
    </div>
</div>
</body>
</html>

```

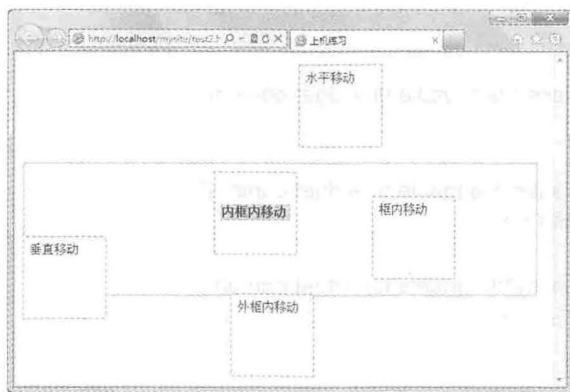


图 13.4 设置移动方向和范围

3. 设置移动对齐方式

下面示例演示了如何为移动元素设置移动对齐方式，对齐方式包括对齐目标，以及内沿或者外沿，也可以指定网格宽度，具体演示请看下面代码。演示效果如图 13.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#draggable" ).draggable({ snap: true });
    $( "#draggable2" ).draggable({ snap: ".ui-widget-header" });
    $( "#draggable3" ).draggable({ snap: ".ui-widget-header", snapMode: "outer" });
    $( "#draggable4" ).draggable({ grid: [ 20,20 ] });
    $( "#draggable5" ).draggable({ grid: [ 80, 80 ] });
});
</script>
<style type="text/css">
.draggable { width: 90px; height: 80px; padding: 5px; float: left; margin: 0 10px 10px 0; font-size: .9em; }
.ui-widget-header p, .ui-widget-content p { margin: 0; }
#snaptarget { height: 140px; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="demo">
    <div id="snaptarget" class="ui-widget-header"></div>
    <div id="draggable" class="draggable ui-widget-content">
        <p>默认效果</p>
    </div>
    <div id="draggable2" class="draggable ui-widget-content">
```

```

        <p>对准大框</p>
    </div>
    <div id="draggable3" class="draggable ui-widget-content">
        <p>对准外沿</p>
    </div>
    <div id="draggable4" class="draggable ui-widget-content">
        <p>对准 20*20 网格</p>
    </div>
    <div id="draggable5" class="draggable ui-widget-content">
        <p>对准 80*80 网格</p>
    </div>
</div>
</body>
</html>

```

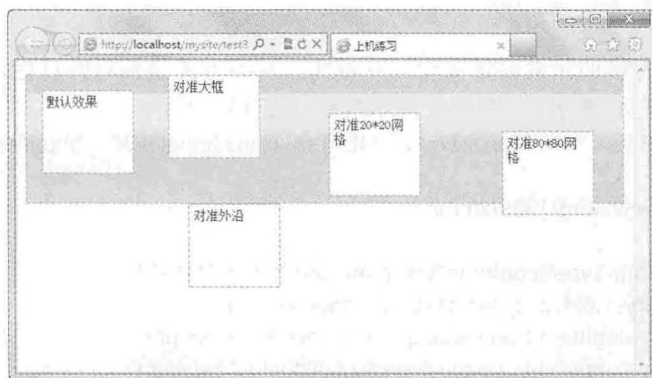


图 13.5 设置移动对齐方式

13.1.2 投放对象

使元素能够被拖动，将会提升 Web 页面的交互性与功能性。接下来就需要为目标元素设置放置目标，此时就要用到投放组件。投放组件为拖动元素提供了可投放的地点，并且在将拖动元素投放到该区域时触发某种操作，使用扩展的事情模型可以很容易对投放事件进行响应。

实现拖动行为需要引入下面几个 JavaScript 文件，然后为拖动目标元素绑定 `droppable()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.draggable.js
- ☒ jquery.ui.droppable.js

【示例 2】 将 `<div id="droppable">` 设置为投放区域，当被拖动的对象放在投放目标区域之后，目标区域将变换背景样式，同时被更新区域内包含的文本内容。演示效果如图 13.6 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>

```

```

<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.droppable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function(){
    $( "#draggable" ).draggable();
    $( "#droppable" ).droppable({
        drop: function( event, ui ) {
            $( this )
                .addClass( "ui-state-highlight" )
                .find( "p" )
                .html( "完成投放" );
        }
    });
})
</script>
<style type="text/css">
#draggable { width: 100px; height: 100px; }
#droppable { width: 150px; height: 150px; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="draggable" class="ui-widget-content">
    <p>拖动目标</p>
</div>
<div id="droppable" class="ui-widget-header">
    <p>投放区域</p>
</div>
</body>
</html>

```

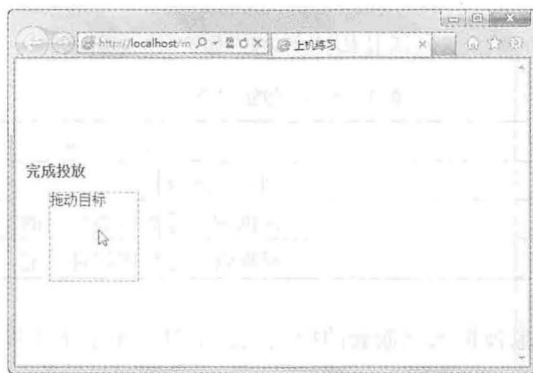


图 13.6 投放区域演示效果

在默认情况下投放组件的实现没有任何操作，它只是在运行的页面中指定可投放的目标，但是除此之外不会发生任何动作，即使在拖动对象放入到该元素中时。这里所说的不发生的事情，指的是并没有添加特定操作代码，在整个交互过程中，拖动对象和投放对象仍然会触发相应的事情。

投放组件具有大量可以配置的属性，使用这些属性，用户能够精确控制拖放行为，说明如表 13.5 所示。

表 13.5 投放组件的属性

属 性	说 明
accept	设置投放对象所能够接收的元素
activeClass	设置在可接收的拖动元素处于拖动状态时, 投放对象的样式类
greedy	当拖放对象被拖入到嵌套的投放元素中时, 用于阻止投放事件被连环调用
hoverClass	指定投放对象在拖动对象被移动到其中时的样式
tolerance	设置所接收的拖动对象被认为完成投放的触发模式

投放对象的 tolerance 属性设置该对象探测拖动对象是否已完成投放的方法, 默认值为 interest, 取值详细说明如表 13.6 所示。

表 13.6 tolerance 属性值

取 值	说 明
fit	只有在拖动对象完全处于投放对象的边界之内时才会认为完成投放
interest	至少 25%的拖动对象进入投入对象边界时才会认为完成了投放
pointer	必须在鼠标指针接触到投放对象边界时才会认为完成投放
touch	只要拖动对象的边缘与投放对象的边缘相接触就会认为完成投放

除了上面介绍的属性外, 投放组件还提供了 5 个事件, 专门用于特定事件执行代码的回调函数, 说明如表 13.7 所示。

表 13.7 投放组件的事件

事 件	说 明
activate	当所接收的拖动对象开始拖动
deactivate	当所接收的拖动对象停止拖动
drop	当所接收的拖动对象被放入到投放对象中
out	当所接收的拖动对象从投放对象内部移出其边界
over	当所接收的拖动对象到投放对象边界之内

投放组件定义了 3 种基本方法, 不包括其构造方法, 如表 13.8 所示。

表 13.8 投放组件的方法

方 法	说 明
destroy()	彻底清除该控件
enable()	根据索引号激活被禁用的组件功能
disable()	根据索引号禁用组件功能

【示例 3】 演示当拖放对象被拖入到嵌套的投放元素中时, 用于阻止投放事件被连环调用对比。演示效果如图 13.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
```

```

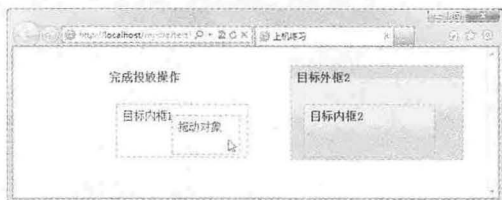
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.droppable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $( "#draggable" ).draggable();
    $( "#droppable, #droppable-inner" ).droppable({
        activeClass: "ui-state-hover",
        hoverClass: "ui-state-active",
        drop: function( event, ui ) {
            $( this )
                .addClass( "ui-state-highlight" )
                .find( "> p" )
                    .html( "完成投放操作" );
            return false;
        }
    });
    $( "#droppable2, #droppable2-inner" ).droppable({
        greedy: true,
        activeClass: "ui-state-hover",
        hoverClass: "ui-state-active",
        drop: function( event, ui ) {
            $( this )
                .addClass( "ui-state-highlight" )
                .find( "> p" )
                    .html( "完成投放操作" );
        }
    });
});
</script>
<style type="text/css">
#draggable { width: 80px; height: 40px; padding: 0.5em; float: left; margin: 10px 10px 10px 0; }
#droppable, #droppable2 { width: 230px; height: 120px; padding: 0.5em; float: left; margin: 10px; }
#droppable-inner, #droppable2-inner { width: 170px; height: 60px; padding: 0.5em; float: left; margin: 10px; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="draggable" class="ui-widget-content">
    <p>拖动对象</p>
</div>
<div id="droppable" class="ui-widget-header">
    <p>目标外框 1</p>
    <div id="droppable-inner" class="ui-widget-header">
        <p>目标内框 1</p>
    </div>
</div>
<div id="droppable2" class="ui-widget-header">
    <p>目标外框 2</p>

```

```

<div id="droppable2-inner" class="ui-widget-header">
  <p>目标内框 2</p>
</div>
</div>
</body>
</html>

```



(a) 连环调用



(b) 禁止连环调用

图 13.7 禁止连环调用前后对比效果

13.2 缩 放

缩放是一种比较灵活的组件，可以广泛地用于其他类型元素，如设计<textarea>标签可能包含不定数量的用户输入文本时可以根据输入文本动态调整文本区域的大小。尺寸改变组件与其他组件能够很好地搭配，并经常与拖动组件一起使用。不过，UI 库可以帮助用户轻易制作可拖动并可以改变尺寸的组件，如对话框，并且这两种组件并不需要任何关联。

实现缩放行为需要引入下面几个 JavaScript 文件，然后为缩放目标元素绑定 `resizable()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.mouse.js
- ☒ jquery.ui.resizable.js

【示例 4】 为一个盒子 (div 元素) 绑定缩放功能，这样当鼠标移动到盒子的右边、底边或者右下角时，可以拖动鼠标实现动态改变元素大小。演示效果如图 13.8 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.resizable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function(){
    $("#resize").resizable();
})
</script>
<style type="text/css">

```

```
#resize { width: 100px; height: 100px; border:solid 1px red; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="resize"> </div>
</body>
</html>
```

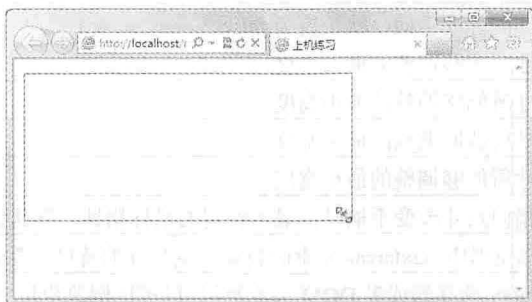


图 13.8 基本缩放效果

如果仅激活拖动功能，只需要引入 UI 库中的下列几个文件：

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.resizable.js
- ☒ jquery.ui.resizable.css

组件将会自动为拖动手柄添加 3 个所需要的元素，甚至会在鼠标指向元素边界时，添加一个指针以提示它可以执行尺寸改变。

使用边界处的手柄可以独立地改变每个方向轴上的尺寸，或者也可以使用角落处的手柄同时改变两轴上的尺寸。上述功能组件已经自动实现这些特性。尽管库主题提供了很多外观漂亮的尺寸改变手柄，但是也可以不用这些主题。如果一个元素尺寸改变，并且没有指定主题，元素将会自动被赋予浅灰色的边框，以提示可以拖曳改变其尺寸。

缩放组件具有大量可以配置的属性，使用这些属性，用户能够精确控制缩放行为，说明如表 13.9 所示。

表 13.9 缩放组件的属性

属 性	说 明
animate	为元素的尺寸改变过程添加动画效果。默认为 false，关闭动画效果
animateDuration	设置动画的速度，可以使用毫秒数作为它的值，也可以使用字符串 show、normal 或者 fast
animateEasing	为尺寸改变动画添加平缓效果。默认值为 swing
alsoResize	在尺寸可变元素改变大小时，使用 jQuery 选择器可以同时改变另一个元素的尺寸。默认值为 false
aspectRatio	使尺寸改变元素的所有边框大小同时改变，以维持元素的高度比。默认值为 false
autoHide	隐藏尺寸改变手柄，直到鼠标指针指向该位置为止。默认值为 false
cancel	使特定元素的尺寸不再可变。默认值为 input
containment	限制尺寸改变元素不能够超出特定容器元素的边界。默认值为 false
delay	设置从指针点击改变手柄到开始改变尺寸的延时毫秒数。默认值为 0
disableSelection	禁止手柄和尺寸改变 helper 元素被选中。默认值为 true
distance	设置在按下鼠标键后，鼠标指针必须移动多少像素后才能够开始改变尺寸。默认值为 1
ghost	在改变尺寸时显示一个替代元素。默认值为 false

续表

属 性	说 明
grid	接收一个对象, 使元素不能够连续地改变尺寸, 只能沿 x 轴或者 y 轴离散的改变长度。默认值为 false
handles	使用对象定义用于改变尺寸的手柄, 其中手柄名称为 (n、s、w 等) 作为键, 而 jQuery 元素选择器或 DOM 节点作为值。默认值为 {e,se,s}
helper	激活 helper 元素以显示尺寸改变的过程, 与 ghost 属性类似, 但更为简单
knobHandles	使用简单的非图片式手柄。默认值为 false
maxHeight	设置元素尺寸所能够调整的最大高度
minHeight	设置元素尺寸所能够调整的最小高度
maxWidth	设置元素尺寸所能够调整的最大宽度
minWidth	设置元素尺寸所能够调整的最小宽度
preserveCursor	在鼠标指针指向尺寸改变手柄时, 显示特定的鼠标指针。默认值为 true
preventDefault	禁止 Safari 浏览器中<textarea>元素的自动改变尺寸的特性。默认值为 true
proportionallyResize	接收一个 jQuery 选择器或者 DOM 节点数组, 以按比例改变尺寸可变元素的大小。默认值为 false
transparent	在交互之前、之中和之后都不显示尺寸改变手柄

除了上面介绍的属性外, 缩放组件还提供了 3 个事件, 专门用于特定事件执行代码的回调函数, 说明如表 13.10 所示。

表 13.10 缩放组件的事件

事 件	说 明
resize	当缩放元素被改变尺寸时执行回调函数
start	当开始拖动时执行回调函数
stop	当改变元素尺寸结束之后, 执行回调函数

缩放组件定义 3 个基本方法, 不包括其构造方法, 如表 13.11 所示。

表 13.11 缩放组件的方法

方 法	说 明
destroy()	彻底清除该控件
enable()	根据索引号激活被禁用的组件功能
disable()	根据索引号禁用组件功能

【示例 5】演示为可改变尺寸元素设置动画效果, 并显示缩放框线。演示效果如图 13.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.resizable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
```

```

<script type="text/javascript" >
$(function() {
    $("#resizable").resizable({
        animate: true
    });
});
</script>
<style type="text/css">
#resizable { width: 150px; height: 150px; padding: 0.5em; }
#resizable h3 { margin: 0; padding: 4px; font-size: 14px; }
.ui-resizable-helper { border: 1px dotted gray; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="resizable" class="ui-widget-content">
    <h3 class="ui-widget-header">国际新闻</h3>
</div>
</body>
</html>

```



图 13.9 缩放栏目

下面设置能够为缩放对象设置最大和最小可缩放范围：

```

$(function() {
    $("#resizable").resizable({
        maxHeight: 250,
        maxWidth: 350,
        minHeight: 150,
        minWidth: 200
    });
});

```

【示例 6】 演示为可改变尺寸元素设置限制范围，缩放区域只能够在这个范围改变。演示效果如图 13.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>

```

```

<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.resizable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $( "#resizable" ).resizable({
        containment: "#container"
    });
});
</script>
<style type="text/css">
#container { width: 300px; height: 300px; }
#container h3 { text-align: center; margin: 0; margin-bottom: 10px; }
#resizable { background-position: top left; width: 150px; height: 150px; }
#resizable, #container { padding: 0.5em; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="container" class="ui-widget-content">
    <h3 class="ui-widget-header">新闻版块</h3>
    <div id="resizable" class="ui-state-active">
        <h3 class="ui-widget-header">国际新闻</h3>
    </div>
</div>
</body>
</html>

```

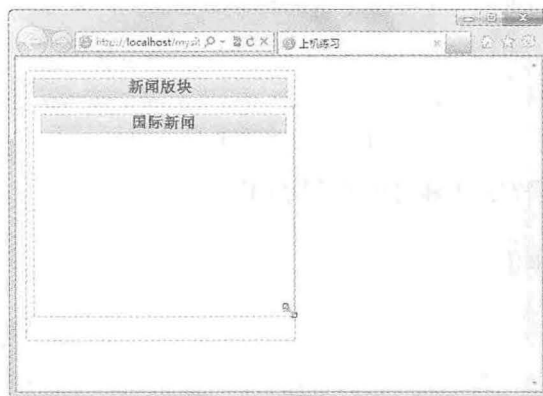


图 13.10 限制缩放范围

13.3 选 择

选择组件用来确定一系列可选的元素，用户只需要简单地拖选元素，或者以单击的方式选择多个元素，行为方式和效果类似于在 Windows 资源管理器中选择多个文件。在现代操作系统中，选择是一种最基本的用户操作行为，jQuery UI 选择组件能够为 Web 页面添加更加直观的操作体验，这对于各种应用场景都是非常有用的。这也是随着 Web 界面的进步，它作为一种应用平台，与桌面应用之间的区别越来越小的一个特征。

实现选择行为需要引入下面几个 JavaScript 文件，然后为选择目标元素绑定 selectable() 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.mouse.js
- ☒ jquery.ui.resizable.js

【示例 7】 使用选择组件为匹配的列表项元素添加选择行为，这样当单击、按住 Ctrl 键单击，或者使用鼠标直接拖选，即可选中多个列表元素。演示效果如图 13.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.selectable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#selectable" ).selectable();
});
</script>
<style type="text/css">
#feedback { font-size: 1.4em; }
#selectable .ui-selecting { background: #FECA40; }
#selectable .ui-selected { background: #F39814; color: white; }
#selectable { list-style-type: none; margin: 0; padding: 0; width: 60%; }
#selectable li { margin: 3px; padding: 0.4em; font-size: 1.4em; height: 18px; }
</style>
<title>上机练习</title>
</head>
<body>
<ol id="selectable">
    <li class="ui-widget-content">列表项目 1</li>
    <li class="ui-widget-content">列表项目 2</li>
    <li class="ui-widget-content">列表项目 3</li>
    <li class="ui-widget-content">列表项目 4</li>
    <li class="ui-widget-content">列表项目 5</li>
    <li class="ui-widget-content">列表项目 6</li>
</ol>
</body>
</html>
```

如果仅激活拖动功能，只需要引入 UI 库中的下列几个文件：

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.resizable.js

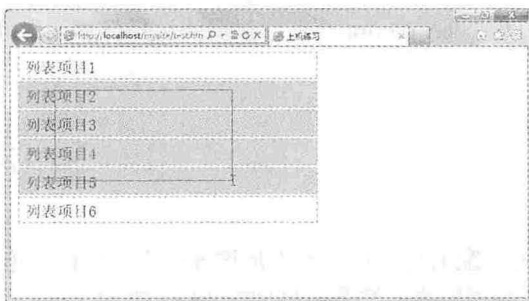


图 13.11 拖选对象

所有被设为可选择的元素，初始时都被设置了一个样式类 `ui-selectee`。当选择方框包含了可选的元素时，它们的样式类被设为 `ui-selecting`。一旦选择交互结束时，任何被选择的元素都被设置为 `selected` 样式，而先前被选过但现在不在所选择范围的元素被设置为 `ui-selecting` 样式。可以很容易地为组件添加自定义样式，以显示元素正在被选择或者已经被选中情况下的外观。

选择组件配置属性比较简洁，只有很少几个配置属性，说明如表 13.12 所示。

表 13.12 选择组件的属性

属 性	说 明
<code>autoRefresh</code>	在开始选择操作之前，自动刷新每个可选项的尺寸和位置。默认值为 <code>true</code>
<code>filter</code>	指定设为可选的子元素，默认值为 <code>*</code>

除了上面介绍的属性外，选择组件还提供了很多事件，专门用于特定事件执行代码的回调函数，说明如表 13.13 所示。

表 13.13 选择组件的事件

事 件	说 明
<code>selected</code>	在选择交互结束时，每个被添加到选项中的元素都会触发一个回调函数
<code>selecting</code>	在选择交互过程中，每个被选中的元素触发此回调函数
<code>start</code>	选择交互开始
<code>stop</code>	在选择交互结束时，触发该属性的回调函数，无论其被选中的项目数有多少个
<code>unselected</code>	在交互过程中任何可被选择却没有被选中的元素都将触发此回调方法
<code>unselecting</code>	不可选择元素在本次选择交互中将会触发此属性

选择组件定义 5 个基本方法，不包括其构造方法，如表 13.14 所示。

表 13.14 选择组件的方法

方 法	说 明
<code>destroy()</code>	彻底清除该控件
<code>enable()</code>	根据索引号激活被禁用的组件功能
<code>disable()</code>	根据索引号禁用组件功能
<code>refresh()</code>	当 <code>autoRefresh</code> 属性值设置为 <code>false</code> ，手动刷新选择组件的位置和样式
<code>toggle()</code>	选择组件的激活和禁用状态切换

该组件具有两个独立的新方法，即 `toggle()` 和 `refresh()` 方法。当 `autoRefresh` 属性值设置为 `false` 时，`refresh()` 方法可用于在特定时刻手动地执行刷新动作。当页面上有许多选择组件时，将 `autoRefresh` 属性值设置为 `false`，可以提高性能。如果这时需要刷新组件的尺寸和位置，那么 `refresh()` 方法就正好派上用场。

toggle()方法可以轻松切换激活和禁用状态,而不需要为这两个状态单独编码,甚至不需要事先侦测当前状态。如果选择组件现在处于激活状态,那么 toggle()方法将会禁用它们。反之如果处于禁用状态,那么 toggle()将会激活它们。该方法的使用方式是非常简单的,但是其他组件并没有这种方法。

【示例 8】 演示如何动态跟踪被选中的项目。演示效果如图 13.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.selectable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#selectable" ).selectable({
        stop: function() {
            var result = $( "#select-result" ).empty();
            $( ".ui-selected", this ).each(function() {
                var index = $( "#selectable li" ).index( this );
                result.append( " 列表项目" + ( index + 1 ) );
            });
        }
    });
});
</script>
<style type="text/css">
#feedback { font-size: 1.4em; }
#selectable .ui-selecting { background: #FECA40; }
#selectable .ui-selected { background: #F39814; color: white; }
#selectable { list-style-type: none; margin: 0; padding: 0; width: 60%; }
#selectable li { margin: 3px; padding: 0.4em; font-size: 1.4em; height: 18px; }
</style>
<title>上机练习</title>
</head>
<body>
<p id="feedback">
<span>选中项目包括:</span> <span id="select-result"></span>
</p>
<ol id="selectable">
<li class="ui-widget-content">列表项目 1</li>
<li class="ui-widget-content">列表项目 2</li>
<li class="ui-widget-content">列表项目 3</li>
<li class="ui-widget-content">列表项目 4</li>
<li class="ui-widget-content">列表项目 5</li>
<li class="ui-widget-content">列表项目 6</li>
</ol>
</body>
</html>
```

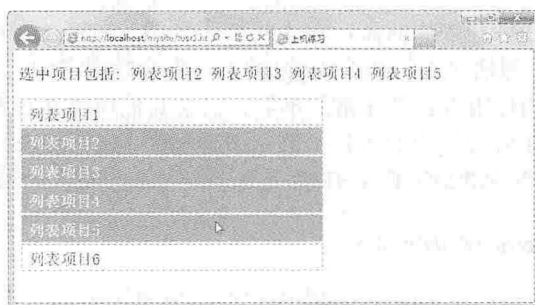


图 13.12 显示被选中的项目

13.4 排 序

排序也是一种交互式行为组件，它可用于一个或者多个元素列表，并且列表中的每个条目可以被重新排序，当然这个列表并不是或者等标签定义的列表结构。排序组件类似于拖放组件的一种专用实现，并具有专门的作用，它包含大量的 API 提供各种使用方式。

实现选择行为需要引入下面几个 JavaScript 文件，然后为选择目标元素绑定 `selectable()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.mouse.js
- ☒ jquery.ui.sortable.js

【示例 9】 使用排序组件为匹配的列表项元素添加手动排序行为，这样当拖动列表项目时，可以重新设置它在当前列表结构中的位置。演示效果如图 13.13 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.sortable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#sortable" ).sortable();
});
</script>
<style type="text/css">
#sortable { list-style-type: none; margin: 0; padding: 0; width: 60%; }
#sortable li { margin: 0 3px 3px 3px; padding: 0.4em; padding-left: 1.5em; font-size: 1.4em; height: 18px; }
#sortable li span { position: absolute; margin-left: -1.3em; }
</style>
```

```

<title>上机练习</title>
</head>
<body>
<ul id="sortable">
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 1</li>
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 2</li>
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 3</li>
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 4</li>
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 5</li>
  <li class="ui-state-default"><span class="ui-icon ui-icon-arrowthick-2-n-s"></span>列表项目 6</li>
</ul>
</body>
</html>

```



图 13.13 拖选对象

如果仅激活拖动功能，只需要引入 UI 库中的下列几个文件：

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.sortable.js

在页面中添加了很多行为，当在列表中向上或者向下拖动某个项目时，其他项目会自动移动位置，以便为当前排序条目提供可投放的空间。当待排序条目被放下时，它将会快速而平稳地滑动到列表中新的位置。

排序组件具有大量的可配置属性，多于其他任何交互组件，但是比日期选择组件要少一些，说明如表 13.15 所示。

表 13.15 排序组件的属性

属 性	说 明
appendTo	设置在排序时需要附加 helper 的元素。默认为 parent
axis	限制排序元素只能够沿着某个方向轴移动，如 x 或者 y 轴。默认为 none
cancel	指定不能够被排序的元素。默认值为:input
connectWith	指定一个排序列表的数组，其中每个列表的排序条目都可以相互移动。默认值为[]
containment	限制排序组件的条目在拖动时不能够超出容器，可用值为字符串 parent、window、document 或者 jQuery 元素选择器
cursor	定义在拖动条目光标的 CSS。默认值为 none
delay	设置从点击（按下鼠标按钮）到开始排序所延迟的毫秒数。默认值为 0
distance	设置在按下鼠标左键后，鼠标指针至少移动多少像素排序操作才能够开始。默认值为 1
dropOnEmpty	允许所关联的条目可以被投放到空位置中，默认值为 true

续表

属 性	说 明
forcePalceholdSize	强制占位符应具有的尺寸。占位符指的是用于放置排序条目的空白位置。默认值为 false
grid	设置排序条目只能够沿着网格拖动，它的值应该为具有两个条目的数组，即网格的 x 和 y 方向的距离。默认值为[]
handle	指定用于拖动排序条目的手柄。默认值为 none
helper	指定助手元素在拖动时的代理元素，可接收返回一个元素的函数作本属性的值。默认值为 original
items	指定可以排序的条目，默认情况下可以为所有子元素排序。默认值为>*
opacity	指定被排序元素的透明度。默认值为 1
placeholder	指定用于放置排序元素的空白位置的 CSS 样式类。默认值为 none
revert	在向新位置移动排序条目时激活动画。默认值为 true
scroll	当排序组件达到可视范围的边缘时让页面可卷页。默认值为 true
scrollSensitivity	设置在卷页开始时排序条目可以靠近可视区域边缘的最小像素值。默认值为 20
scrollSpell	设置当发生卷页时，可视区所卷动的距离像素数，默认值为 20
zIndex	排序条目或者助手元素在拖动时的 z-index 值

除了上面介绍的属性外，排序组件还提供了很多事件，它们可以接收一些函数作为事件处理函数，并在排序交互中的不同时间点执行，说明如表 13.16 所示。

表 13.16 排序组件的事件

事 件	说 明
activate	在连接列表间开始排序时执行回调函数
beforeStop	当排序结束但原来的位置仍然可用时执行回调函数
change	排序过程中，排序元素的 DOM 位置发生改变时执行回调函数
deactive	当连接列表间的排序停止时执行回调函数
out	当待排序元素从连接列表中移出时执行回调函数
over	当连接列表间的排序完成时执行回调函数
receive	当从所连接的列表中接收待排序条目时执行回调函数
remove	当在连接列表的排序条目移动时执行回调函数
sort	当排序发生时执行回调函数
start	当排序开始时执行回调函数
stop	当排序结束时执行回调函数
update	当排序结束后，且 DOM 位置也发生了变化时执行回调函数

这些事件处理器十分重要，因为它们能够在特定情况发生时进行响应。这些事件大多数在单个排序交互中触发，它们被触发的顺序为 start→sort→change→beforeStop→stop→update。

一旦排序发生时，start 事件将首先被触发，之后每次鼠标移动都会触发 sort 事件，这使得该事件的发生非常密集。当另一个条目被当前待排序条目替换时，change 事件将会被触发。一旦排序条目被放下时，beforeStop 和 stop 事件将相继被触发。最后，如果排序条目处于新的位置，将会触发 update 事件。

如果至少一个条目在连接列表之间发生了移动，那么触发这些事件的顺序为 start→activate→sort→change→beforeStop→stop→remove→update→receive→reactivate。

排序组件开放了通用的函数以使组件能够执行某些操作，并且和前面介绍的选择组件一样，排序组件也定义了一组其他组件没有的专有方法，如表 13.17 所示。

表 13.17 排序组件的方法

方 法	说 明
destroy()	彻底清除该控件
enable()	根据索引号激活被禁用的组件功能
disable()	根据索引号禁用组件功能
refresh()	触发刷新排序条目集合
refreshPosition()	触发刷新排序条目集合的缓存
serialize()	构造 URL 附加字符串以向服务器发送新的排序次序
toArray()	将排序条目序列化转换为字符串

这些方法大多数在之前使用过，下面简单介绍最后几种方法的使用。

refresh()和 refreshPosition()方法类似，但是 refreshPosition()方法将刷新排序条目的缓存位置。该方法由组件在适当的时候自动调用，也可以在需要时手动调用。但是在对该方法使用时，应该有限度，因为它对页面的影响比较强烈。serialize()方法比较重要，它用于在排序结束后对结果进行处理，即将页面上的元素格式化为简单的字符串，以便通过网络传送到服务器端应用。

【示例 10】 演示如何通过鼠标拖曳来调整两列栏目之间的相互动态排序。演示效果如图 13.14 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.sortable.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $(".column").sortable({
        connectWith: ".column"
    });
    $(".portlet").addClass("ui-widget ui-widget-content ui-helper-clearfix ui-corner-all")
        .find(".portlet-header")
            .addClass("ui-widget-header ui-corner-all")
            .prepend("<span class='ui-icon ui-icon-minusthick'></span>")
            .end()
        .find(".portlet-content");
    $(".portlet-header .ui-icon").click(function() {
        $(this).toggleClass("ui-icon-minusthick").toggleClass("ui-icon-plusthick");
        $(this).parents(".portlet:first").find(".portlet-content").toggle();
    });
    $(".column").disableSelection();
});
</script>
<style type="text/css">
```

```

.column { width: 170px; float: left; padding-bottom: 100px; }
.portlet { margin: 0 1em 1em 0; }
.portlet-header { margin: 0.3em; line-height: 2em;; padding-left: 0.2em; }
.portlet-header .ui-icon { float: right; }
.portlet-content { padding: 0.4em; }
.ui-sortable-placeholder { border: 1px dotted black; visibility: visible !important; height: 50px !important; }
.ui-sortable-placeholder * { visibility: hidden; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="column">
  <div class="portlet">
    <div class="portlet-header">左栏 1</div>
    <div class="portlet-content">内容</div>
  </div>
  <div class="portlet">
    <div class="portlet-header">左栏 2</div>
    <div class="portlet-content">内容</div>
  </div>
</div>
<div class="column">
  <div class="portlet">
    <div class="portlet-header">右栏 1</div>
    <div class="portlet-content">内容</div>
  </div>
</div>
</body>
</html>

```

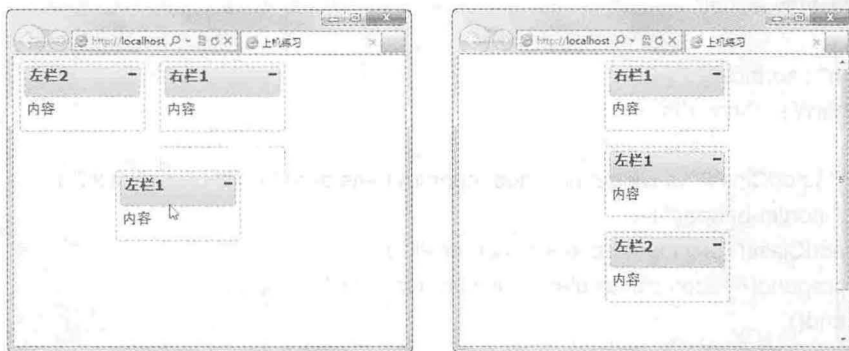



图 13.14 动态改变栏目位置

第14章

jQuery UI 部件开发

( 视频讲解：51 分钟)

使用 jQuery 可以构建良好的 Web 应用程序所需要的核心框架，但设计强大的 Web 应用程序，如果从零开始进行设计，整个开发过程依然很艰巨，用户最需要的是通过几个步骤修改现有的应用程序，让它能够在各种场合中顺利运行，并且适合所有用户。部件开发是一组界面视图，通过简单的调用，就可以快速实现各种常规 Web 应用的设计。jQuery UI 部件组件包括 8 种，每种都适用于特定的应用，说明如图 14.1 所示。

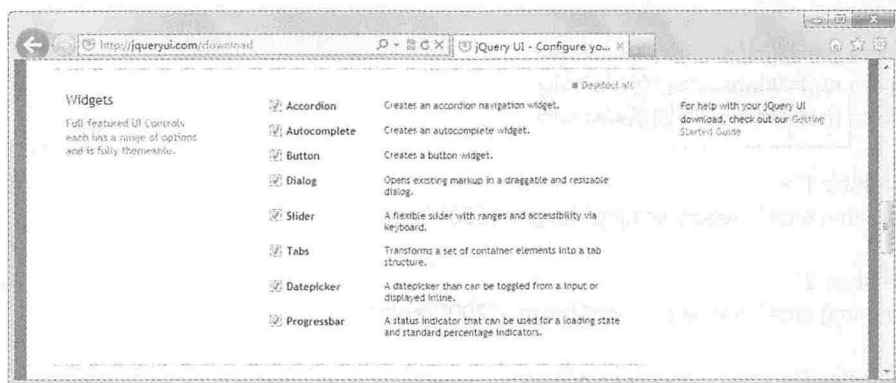


图 14.1 jQuery UI Widgets

14.1 选项卡

选项卡组件用于在一组不同元素之间切换视角，可以通过单击每个元素的标题来访问该元素包含的内容，这些标题都作为独立的选项卡而出现。每个元素，或者说每个内容片断都具有一个与之关联的选项卡，并且在同一时刻只能够打开其中一个内容片断。

添加选项卡部件需要在页面中引入下面几个 JavaScript 文件，然后为目标选项卡包含框绑定 `tabs()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.tabs.js

☑ jquery.ui.all.css

【示例 1】在网页中插入一个选项卡面板，不需要任何设置和编程。演示效果如图 14.2 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.tabs.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#tabs" ).tabs();
});
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
<div id="tabs">
    <ul>
        <li><a href="#tabs-1">新闻</a></li>
        <li><a href="#tabs-2">社区</a></li>
        <li><a href="#tabs-3">联系</a></li>
    </ul>
    <div id="tabs-1">
        <p></p>
    </div>
    <div id="tabs-2">
        <p></p>
    </div>
    <div id="tabs-3">
        <p></p>
    </div>
</div>
</body>
</html>
```



图 14.2 基本选项卡效果

选项卡组件是基于底层的 HTML 元素结构，该结构是固定的，组件的运转依赖一些特定的元素。选项卡本身必须从列表元素中创建，列表结构可以是排序的，也可以是无序的，并且每个列表项应当包含一个 span 元素和一个 a 元素。每个链接还必须具有相应的 div 元素，与它的 href 属性相关联。例如：

```
<ul>
  <li><a href="#tabs"><span>标题</span></a></li>
</ul>
<div id="tabs1">Tab 面板容器 </div>
```

对于该组件来说，必要的 CSS 样式是必需的，默认可以导入 jquery.ui.all.css 文件或者 jquery.ui.tabs.css，也可以自定义 CSS 样式表，用来控制选项卡的基本样式。

一套选项卡面板包括以下几种以特定方式排列的标准 HTML 元素，根据实际需要可以在页面中编写好，也可以动态添加，或者两者结合。

- ☒ 列表元素 (ul 或 ol)
- ☒ a 元素
- ☒ span 元素
- ☒ div 元素

前 3 个元素组成了可单击的选项标题，以用来打开选项卡所关联的内容框，每个选项卡应该包含一个带有链接的列表项，并且链接内部还应嵌套一个 span 元素。每个选项卡的内容通过 div 元素创建，其 id 值是必需的，标记了相应的 a 元素的链接目标。

jquery.ui.all.css 或 jquery.ui.tabs.css 样式表文件包含了所有能够保证选项卡组件外观和功能的样式，用户可以提供自己的样式表，只要其中包含了所需要的规则选项，也可以通过使用主题定制选项卡的主题。

选项卡组件具有大量可以配置的属性，使用这些属性，用户能够精确控制选项卡的功能和外观，说明如表 14.1 所示。

表 14.1 选项卡组件属性

属 性	说 明
ajaxOptions	远程 Ajax 选项卡选项，默认值为 {}
cache	只载入远程选项卡内容一次，延迟加载，默认值为 false
cookie	在页面载入时利用 cookie 数据显示激活的选项卡。默认值为 null
disabled	在页面载入时禁用特定的选项卡。默认值为 []
idprefix	在远程选项卡链接元素没有 title 属性时使用。默认值为 ui-tabs
event	触发器显示内容时的选项卡事件。默认值为 click
fx	指定选择选项卡时的切换效果，默认为 null
panelTemplate	对于动态创建选项卡的内容包含框，本字符串描述了其使用的元素。默认为<div></div>
selected	当组件被渲染时，默认选中的选项卡，默认值为 0
spinner	指定远程选项卡载入时的等待标记字符串，默认值为Loading…
tabTemplate	用于描述动态创建选项卡时所使用的元素字符串，默认值为 '#{label}'
unselect	在单击以选中的选项卡时将其隐藏

除了上面介绍的属性外，选项卡组件还提供了很多事件，专门用于特定事件执行代码的回调函数，说明如表 14.2 所示。

表 14.2 选项卡组件事件

事 件	说 明
add	在添加新选项卡时执行的回调函数

续表

事 件	说 明
disable	在禁用选项卡时执行的回调函数
enable	在激活选项卡时执行的回调函数
load	在选项卡远程载入了数据时执行的回调函数
remove	在选项卡被移除时执行的回调函数
select	在选项卡被选中时执行的回调函数
show	在选项卡的内容被显示时执行的回调函数

选项卡组件定义多个基本方法，不包括其构造方法，如表 14.3 所示。

表 14.3 选项卡方法

方 法	说 明
add()	通过编程方式添加选项卡，并在参数中指明选项卡内容的 url、label 和索引号（可选的）
remove()	通过编程方式移除选项卡，需要指明要去除的选项卡索引号
enable()	根据索引号激活禁用的选项卡
disable()	根据索引号禁用选项卡
select()	通过编程方式根据索引号选中相应 un，其效果与访问者单击选项卡相同
load()	选项卡被禁用
url()	改变 Ajax 选项卡内容的 URL，该方法需要指明选项卡的索引号和新的 URL
destroy()	彻底去除整个选项卡
length()	组件中选项卡的数量
rotate()	一次性或重复地在经过指定的毫秒数之后，自动改变组件中活动的选项卡

1. 设计可折叠的选项卡

下面示例演示了如何调用选项卡组件，并设置 collapsible 属性值为 true，设计可折叠的选项卡，当单击当前选项卡时可以展开或者折叠。演示效果如图 14.3 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.tabs.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#tabs" ).tabs({
        collapsible: true
    });
});
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
```

```

<body>
<div id="tabs">
  <ul>
    <li><a href="#tabs-1">新闻</a></li>
    <li><a href="#tabs-2">社区</a></li>
    <li><a href="#tabs-3">联系</a></li>
  </ul>
  <div id="tabs-1">
    <p></p>
  </div>
  <div id="tabs-2">
    <p></p>
  </div>
  <div id="tabs-3">
    <p></p>
  </div>
</div>
</body>
</html>

```



图 14.3 设计可折叠的选项卡

2. 重新布局选项卡

下面示例演示了如何把选项卡的标题栏放在面板的底部，要实现该效果，需要适当添加额外的 CSS 样式，用来定位选项卡标题栏，并绑定一行代码，添加和移出控制类样式。演示效果如图 14.4 所示。当把标题栏放置到面板底部时，必须明确设置面板的高度。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.tabs.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
  $("#tabs").tabs();
  $(".tabs-bottom .ui-tabs-nav, .tabs-bottom .ui-tabs-nav > *")
    .removeClass("ui-corner-all ui-corner-top")

```



```

        .addClass( "ui-corner-bottom" );
    });
</script>
<style type="text/css">
#tabs { height: 300px; }
.tabs-bottom { position: relative; }
.tabs-bottom .ui-tabs-panel { height: 240px; overflow: auto; }
.tabs-bottom .ui-tabs-nav { position: absolute !important; left: 0; bottom: 0; right: 0; padding: 0 0.2em 0.2em 0; }
.tabs-bottom .ui-tabs-nav li { margin-top: -2px !important; margin-bottom: 1px !important; border-top: none;
border-bottom-width: 1px; }
.ui-tabs-selected { margin-top: -3px !important; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="tabs" class="tabs-bottom">
    <ul>
        <li><a href="#tabs-1">新闻</a></li>
        <li><a href="#tabs-2">社区</a></li>
        <li><a href="#tabs-3">联系</a></li>
    </ul>
    <div id="tabs-1">
        <p></p>
    </div>
    <div id="tabs-2">
        <p></p>
    </div>
    <div id="tabs-3">
        <p></p>
    </div>
</div>
</body>
</html>

```

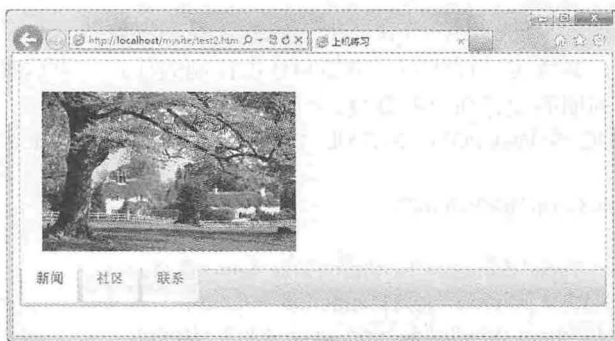


图 14.4 重新布局选项卡标题栏位置

14.2 手 风 琴

手风琴组件是另一种由一系列内容容器所组成的组件，这些容器在同一个时刻只能有一个被打开。

因此, 初始化使它的大多数内容在界面上是被隐藏的, 这与 14.1 节介绍的选项卡组件非常相似。

每个容器都有一个与之关联的标题元素, 用来打开该容器并显示其内容。当单击某个容器的标题时, 它的内容将会被展示出来。当单击另一个标题时, 当前可见的内容将会被隐藏, 而新的内容将会被显示在页面中。

手风琴组件是强大而高度可配置的组件, 它通过在同一时间只显示相关内容中的一部分来节省 Web 页面空间, 这与选项卡是类似的, 只不过它是垂直摆放而不是水平摆放的。

手风琴组件不仅对于页面访问者来说易于使用, 对于开发者来说也是易于实现的, 它具有一系列用来定制外观和行为的可配置属性。此外, 还包含一组方法, 能够用来以编程方式控制组件。该组件中容器元素的高度是可自动设置的, 以确保除了显示标题之外, 还有足够的空间显示最大高度的内容面板。

添加选项卡部件需要在页面中引入下面几个 JavaScript 文件, 然后为目标手风琴包含框绑定 accordion() 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.accordion.js
- ☒ jquery.ui.all.css

【示例 2】 在网页中插入一个手风琴面板, 不需要任何设置和编程。演示效果如图 14.5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.accordion.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#accordion" ).accordion();
});
</script>
<style type="text/css"></style>
<title>上机练习</title>
</head>
<body>
<div id="accordion">
    <h3><a href="#">新闻</a></h3>
    <div>
        <p></p>
    </div>
    <h3><a href="#">社区</a></h3>
    <div>
        <p></p>
    </div>
    <h3><a href="#">博客</a></h3>
    <div>
        <p></p>
    </div>
</div>
```

```

</div>
<h3><a href="#">联系</a></h3>
<div>
  <p></p>
</div>
</div>
</body>
</html>

```



图 14.5 基本手风琴效果

创建手风琴组件不需要特殊的结构，使用简单的 ID 选择器来指定页面上需要转换为手风琴的元素，然后使用 `accordion()` 构造函数来创建手风琴组件即可。该组件甚至不需要建立在任何列表元素之上，仅使用嵌套的 `div` 和 `a` 元素，就可以构建功能完好的手风琴组件，当然可能需要一些额外的配置。

如果不指定样式，手风琴组件面板将会占据所在容器的 100% 宽度，与其他组件一样，有几种不同的设置样式的方式，可以通过创建自定义的样式表来控制手风琴及其内容的外观，还可以使用 UI 库所提供的 `default` 或 `flora` 主题，或者使用主题定制器创建整个库的扩展皮肤。

手风琴组件具有大量可以配置的属性，使用这些属性，用户能够精确控制手风琴的功能和外观，说明如表 14.4 所示。

表 14.4 手风琴组件属性

属 性	说 明
<code>active</code>	选择初始时打开的抽屉。默认值为 <code>first child</code>
<code>alwaysOpen</code>	保证始终会有一个抽屉被打开。默认值为 <code>true</code>
<code>animated</code>	打开抽屉的动画。默认值为 <code>slide</code>
<code>autoHeight</code>	根据最大的抽屉自动设置高度。默认值为 <code>true</code>
<code>clearStyle</code>	在动画效果之后清除样式。默认值为 <code>false</code>
<code>event</code>	标题事件，以触发打开抽屉。默认值为 <code>click</code>
<code>fillSpace</code>	手风琴完全占据所有容器的高度。默认值为 <code>false</code>
<code>header</code>	选择标题元素。默认值为 <code>a</code>
<code>navigation</code>	激活手风琴导航。默认值为 <code>false</code>
<code>navigationFilter</code>	默认情况下该属性打开 <code>href</code> 属性与 <code>location.href</code> 相匹配的标题所对应的抽。默认值为 <code>location.href</code>
<code>selectedClass</code>	应用于已打开抽屉的样式类名。默认值为 <code>selected</code>

手风琴定义了可定制的 `change` 事件，此事件在打开或者关闭手风琴中的抽屉时触发。为了响应该事件，可以使用可配置属性 `change` 来制定每当事件发生时所需要执行的函数。

手风琴组件包含一系列可供选择的方法，使用它们能够以编程方式控制或者操纵组件的行为。其中一

些方法对于 UI 库中的每个组件都有用, 如 `destroy()` 方法, 该方法用于注销组件。`enable()` 用来激活组件, 而 `disable()` 方法用来禁用组件。

1. 设计带有图标的手风琴

下面示例演示了如何使用手风琴折叠面板定义图标。演示效果如图 14.6 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.accordion.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    var icons = {
        header: "ui-icon-circle-arrow-e",
        headerSelected: "ui-icon-circle-arrow-s"
    };
    $( "#accordion" ).accordion({
        icons: icons
    });
});
</script>
<style type="text/css">
#accordion h3 { height:26px; margin:0; padding:0;}
</style>
<title>上机练习</title>
</head>
<body>
<div id="accordion">
    <h3><a href="#">新闻</a></h3>
    <div>
        <p></p>
    </div>
    <h3><a href="#">社区</a></h3>
    <div>
        <p></p>
    </div>
    <h3><a href="#">博客</a></h3>
    <div>
        <p></p>
    </div>
    <h3><a href="#">联系</a></h3>
    <div>
        <p></p>
    </div>
</div>
</body>
</html>
```

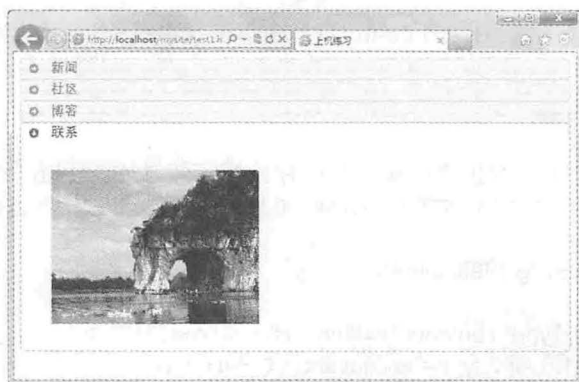



图 14.6 设计带有图标的手风琴效果

2. 折叠手风琴

下面示例演示了自动折叠当前打开的手风琴，只需要设置 `collapsible: true` 即可。详细代码如下，演示效果如图 14.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.accordion.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#accordion" ).accordion({
        collapsible: true
    });
});
</script>
<style type="text/css">
#accordion h3 { height:26px; margin:0; padding:0;}
</style>
<title>上机练习</title>
</head>
<body>
<div id="accordion">
<h3><a href="#">新闻</a></h3>
<div>
<p></p>
</div>
<h3><a href="#">社区</a></h3>
<div>
<p></p>
</div>
<h3><a href="#">博客</a></h3>
<div>
```

```

<p></p>
</div>
<h3><a href="#">联系</a></h3>
<div>
  <p></p>
</div>
</div>
</body>
</html>

```

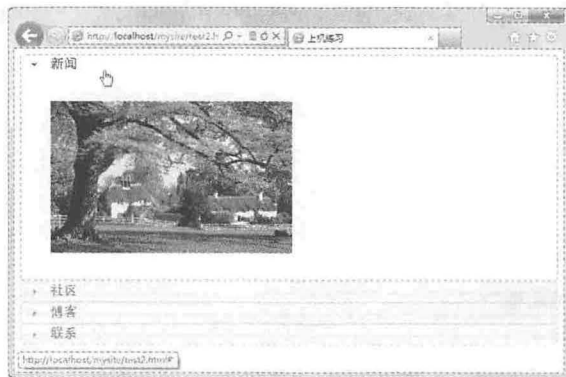


图 14.7 自动折叠的手风琴面板

通过下面代码可以设置手风琴响应事件为鼠标经过，而不是默认的单击事件：

```

$(function() {
  $("#accordion").accordion({
    event: "mouseover"
  });
});

```

14.3 对话框

如果需要在 Web 应用中显示简短的信息提示，或者向访问者询问，可以有两种方式：一是使用 JavaScript 原生的对话框，如 `alert()` 或者 `confirm()` 方法等，另一种方法是打开一个新的页面，预先定义好尺寸，并且将其样式设置为对话框风格。不过，JavaScript 提供的原生方法既不灵活，也不巧妙，它们在解决一个问题的同时，通常会产生新的问题。

jQuery UI 提供了更多功能和更加丰富特性的对话框组件，该对话框组件可以显示消息，附加内容（如图片或文字等），甚至包括交互型内容（如表单），为对话框添加按钮也更加容易，如简单地确定和取消按钮，并且可以为这些按钮定义回调函数，以便在它们被点击时做出反应。

添加对话框部件需要在页面中引入下面几个 JavaScript 文件，然后为目标对话框包含框绑定 `dialog()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.dialog.js
- ☒ jquery.ui.all.css

【示例 3】 在网页中把 `<div id="dialog">` 标签转换为对话框显示。演示效果如图 14.8 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.dialog.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("#dialog").dialog();
});
</script>
<title>上机练习</title>
</head>
<body>
<div id="dialog" title="基本对话框">
    <p>对话框包含内容</p>
</div>
</body>
</html>

```

对于对话框组件来说, 根据需要可以导入下面几个文件。

- ☒ jquery.ui.position.js: 定位对话框。
- ☒ jquery.ui.resizable.js: 调整对话框大小。
- ☒ jquery.ui.draggable.js: 拖动对话框。

这些 JavaScript 文件的相关性比较低, 并且只是在要求对话框尺寸可变或者可拖动的情况下才需要它们, 没有这些文件组件仍然可以运行。对于该对话框来说, jquery.ui.dialog.css 样式表是必需的, 当然可以导入其他主题或者自定义样式表。

除此之外, 对话框组件的初始化方法与已经学习过的其他组件一致, 当在浏览器中运行该页面时, 将会看到默认的对话框。

对话框组件带有内建模式, 在默认情况下是非激活的。而一旦模式被激活, 将会启用一个模式覆盖层元素, 覆盖对话框的父页面。而对话框将会位于该覆盖层的上面, 同时页面的其他部分将位于覆盖层的下面。

这个特性的好处是可以确保对话框被关闭之前, 父页面不能够进行交互, 并且为要求访问者在进一步操作前必须关闭对话框提供了一个清晰的视觉指标。

改变对话框的皮肤使之与内容相适应是很容易的, 可以从默认的主题样式表 (jquery.ui.dialog.css) 中进行修改, 也可以自定义对话框样式表。

对话框组件具有大量可以配置的属性, 使用这些属性, 用户能够精确控制对话框功能和外观, 说明如表 14.5 所示。



图 14.8 基本对话框效果

表 14.5 对话框组件的属性

属 性	说 明
autoOpen	在调用对话框时自动显示对话框。默认值为 true
bgiframe	创建 iframe 片段以防止在 IE 6 中 select 元素穿过对话框显示。现在该功能需要 bgiframe 插件, 但在将来版本中该问题就解决了。默认值为 true

续表

属 性	说 明
buttons	为对话框提供包含按钮的对象。默认值为 {}
dialogClass	为对话框设置额外的样式类名，以指定主题。默认值为 ui-dialog
draggable	让对话框可以拖动，默认值为 true，需要导入 jquery.ui.draggable.js 文件
height	设置对话框的初始高度。默认值为 200px
hide	设置对话框被关闭时使用的效果。默认值为 none
maxHeight	设置对话框的最大高度。默认值为 none
maxWidth	设置对话框的最大宽度。默认值为 none
minHeight	设置对话框的最小高度。默认值为 100px
minWidth	设置对话框的最小宽度。默认值为 150px
modal	在对话框打开时激活其模式。默认值为 false
overlay	带有 CSS 属性的对象，用于模态覆盖层。默认值为 {}
position	设置对话框在视角中的起始位置。默认值为 center
resizable	使对话框大小可改变，需要导入 jquery.ui.resizable.js 文件。默认值为 true
show	设置对话框被打开时使用的效果。默认值为 none
stack	在同时打开几个对话框时，将带有焦点的对话框移到最前面。默认值为 true
title	在对话框源元素中指定标题的替代方法。默认值为 none
width	设置对话框的初始宽度，默认值为 300px

对话框组件有很多可配置的属性，这些属性大多数是布尔型，或者是简单的字符串型，因此很容易设置和使用。在上面示例中，对话框在页面载入时就被打开，当然可以通过 `autoOpen` 属性改变这种行为，使其在某些其他事件发生时，才开始打开对话框。`position` 属性控制对话框被打开时出现在视角中的位置，它接收字符串或者数组值为参数。可以使用的字符串包括以下几种：

- ☒ bottom
- ☒ center
- ☒ left
- ☒ right
- ☒ top

可以用一个数组来精确指定对话框出现时距离左上角的坐标，该坐标指定了在视角中对话框离左上角是偏移距离。

尽管可以在底层为 HTML 元素设置 `title` 属性，来定义对话框的标题栏标题，但是使用组件的 `title` 属性动态定义是首选，因为这样就能够阻止对话框在主体被覆盖时显示标题。

对话框的重要特性就是模态，它在组件被打开时创建一个覆盖层，位于页面内容之上，对话框界面之下，该覆盖层在对话框被关闭时将自动删除，但是在对话框处于被打开状态时，底层页面将不能够进行任何交互。

除了上面介绍的属性外，对话框组件还提供了很多个事件，专门用于特定事件执行代码的回调函数，说明如表 14.6 所示。

表 14.6 对话框组件的事件

事 件	说 明
close	对话框被关闭时执行的回调函数
drag	对话框被拖动时执行的回调函数
dragStart	对话框开始拖动时执行的回调函数

续表

事 件	说 明
dragStop	对话框拖动结束时执行的回调函数
focus	对话框获取焦点时执行的回调函数
open	对话框被打开时执行的回调函数
resize	对话框尺寸被改变时执行的回调函数
resizeStart	对话框尺寸开始改变时执行的回调函数
resizeStop	对话框尺寸改变结束时执行的回调函数

对话框需要一些方法以实现其功能，作为开发者可以轻松打开、关闭或者注销对话框，对话框组件定义多个基本方法，不包括其构造方法，如表 14.7 所示。

表 14.7 对话框组件的方法

方 法	说 明
close()	关闭或者隐藏对话框
destroy()	永久性禁用该对话框
isOpen()	确定对话框是否被打开
moveToTop()	将指定对话框置于最顶层
open()	打开对话框

1. 动态控制对话框显示

下面示例演示了如何通过按钮控制对话框的显示。演示效果如图 14.9 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.dialog.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.position.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.resizable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("#dialog").dialog({
        autoOpen: false,
        show: "blind",
        hide: "explode"
    });
    $("#opener").click(function() {
        $("#dialog").dialog("open");
        return false;
    });
});
</script>
```

```

<title>上机练习</title>
</head>
<body>
<button id="opener">打开对话框</button>
<div id="dialog" title="基本对话框">
  <p>对话框包含内容</p>
</div>
</body>
</html>

```



图 14.9 动态打开对话框

2. 为对话框添加按钮和覆盖层

下面示例演示了为对话框绑定关闭对话框的按钮，并打开覆盖层。演示效果如图 14.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.dialog.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.position.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.resizable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.draggable.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
  $("#dialog").dialog({
    modal: true,
    buttons: {
      Ok: function() {
        $(this).dialog("close")
      }
    }
  });
});
</script>
<style type="text/css">
</style>
<title>上机练习</title>

```

```

</head>
<body>
<div id="dialog" title="基本对话框">
  <p>对话框包含内容</p>
</div>

</body>
</html>

```



图 14.10 启动覆盖层和添加按钮

14.4 滑 动 条

滑动条组件能够实现一种美观且易于使用的界面，它能够为访问者带来直观而有吸引力的使用感受。滑动条的基本功能很简单，它的背景条代表了系列值，而通过移动背景条上的指针可以选择所需要的值。

滑动条由两个主要元素组成，即滑动手柄和滑动条轨道，因此该组件所需要的 HTML 元素是内嵌了 div 元素的 a 元素，此外并没有动态产生任何元素。

添加滑动条部件需要在页面中引入下面几个 JavaScript 文件，然后为目标滑动条包含框绑定 slider() 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.widget.js
- ☒ jquery.ui.mouse.js
- ☒ jquery.ui.all.css

【示例 4】 在网页中把 <div id="slider"> 标签转换为滑动条显示。演示效果如图 14.11 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.slider.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">

```

```
$(function() {  
    $( "#slider" ).slider();  
});  
</script>  
<title>上机练习</title>  
</head>  
<body>  
<div id="slider"></div>  
</body>  
</html>
```



图 14.11 基本滑动条效果

基本滑动条的默认行为是非常简单的，使用鼠标拖动滑动条上的小滑块，指针将会立刻移动到所指向的位置，而一旦指针被选中，则可以通过键盘上的左右方向键移动它。通过改写定义滑动条背景和指针的选择器（jquery.ui.slider.css），可以很容易地改变滑动条的样式。

由于滑动条还没有被配置，新的滑动条暂时还不能够做任何事情。滑动条具有内建特性，它能够自动侦测所要实现的组件是水平还是垂直的。如果想创建垂直滑动条，只需要使用一些自定义图片，并修改一些 CSS 规则即可，滑动条组件会自动完成其他工作。

滑动条组件具有大量可以配置的属性，使用这些属性，用户能够精确控制滑动条功能和外观，说明如表 14.8 所示。

表 14.8 滑动条组件的属性

属 性	说 明
animate	在单击轨道时，为滑动条指针的移动激活平滑效果的动画。默认值为 false
axis	设置滑动条的方向，如果自动侦测失败
handle	设置滑动条指针的样式名。默认值为 ui-slider-handle
handles	设置滑动条指针的边界。默认值为 {}
max	设置滑动条的最大值。默认值为 100
min	设置滑动条的最小值。默认值为 0
range	在两个滑动条之间创建带有样式的区域。默认值为 false
startValue	设置滑动条指针的开始值。默认值为 0
stepping	设置每步之间的距离值
steps	设置步数

stepping 和 steps 属性在使用上是非常相似的，但是不能把它们混淆了。stepping 指的是每步之间的距离，即指针每跳移动的长度。而 steps 指的是步数，而不是它们之间的距离。这两个属性不应该在同一个实现中一起使用。

startValue 属性也是同样易于使用的，根据滑动条要代表的事物，指针的开始值可能不总是为 0。如果需要让指针初始时位于轨道的中间而不是开头，设置 startValue:50 即可。

除了上面介绍的属性外，滑动条组件还提供了很多个事件，专门用于特定事件执行代码的回调函数，

说明如表 14.9 所示。

表 14.9 滑动条组件的事件

事 件	说 明
change	在滑动条指针停止移动并且它的值发生改变时被调用
slide	在滑动条指针移动时被调用
start	在滑动条指针开始移动时被调用
stop	在滑动条指针停止移动时被调用

上面事件回调函数调用顺序为 start→slide→stop→change。

这些回调函数能够根据访问者对滑动条的交互进行响应，而不是仅在页面上显示一条消息。

滑动条是直观而易用的，但是如果获得超出之前所展示的效果，那么就需要它的内建方法，如表 14.10 所示。

表 14.10 滑动条方法

方 法	说 明
moveTo()	将指针移动到轨道的指定值
value()	获取指针的当前值
disable()	禁用滑动条功能
enable()	激活滑动条功能
destroy()	将底层标记返回到原始状态

1. 读写滑动条的值

下面示例演示了如何设置滑动条的值，以及如何设置滑动条的取值范围，并获取当前滑块的值。演示效果如图 14.12 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.slider.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#slider" ).slider({
        range: "min",
        value: 50,
        min: 1,
        max: 120,
        slide: function( event, ui ) {
            $( "#value" ).val( ui.value );
        }
    });
    $( "#value" ).val( $( "#slider" ).slider( "value" ) );
});
```

```

</script>
<title>上机练习</title>
</head>
<body>
<p>
  <label for="value">当前值:</label>
  <input type="text" id="value" />
</p>
<div id="slider"></div>
</body>
</html>

```

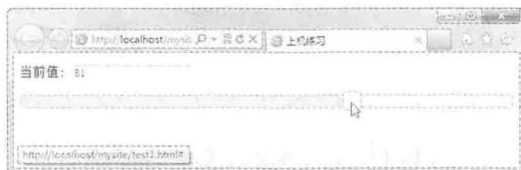


图 14.12 读写滑动条的值

2. 设计垂直双滑动的滑动条

下面示例演示了设置滑动条垂直显示，并定义两个滑块，分别用来控制最大值和最小值。演示效果如图 14.13 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.widget.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.mouse.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.slider.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
  $( "#slider" ).slider({
    orientation: "vertical",
    range: true,
    values: [ 17, 67 ],
    slide: function( event, ui ) {
      $( "#value" ).val( ui.values[ 0 ] + " - " + ui.values[ 1 ] );
    }
  });
  $( "#value" ).val( $( "#slider" ).slider( "values", 0 ) + " - " + $( "#slider" ).slider( "values", 1 ) );
});
</script>
<title>上机练习</title>
</head>
<body>
<p>
  <label for="value">当前值:</label>
  <input type="text" id="value" />

```

```

</p>
<div id="slider" style="height:250px;"></div>
</body>
</html>

```



```

$(function() {
    $( "#red, #green, #blue" ).slider({
        orientation: "horizontal",
        range: "min",
        max: 255,
        value: 127,
        slide: refreshSwatch,
        change: refreshSwatch
    });
    $( "#red" ).slider( "value", 255 );
    $( "#green" ).slider( "value", 140 );
    $( "#blue" ).slider( "value", 60 );
});
</script>
<style type="text/css">
#red, #green, #blue { float: left; clear: left; width: 300px; margin: 15px; }
#swatch { width: 120px; height: 140px; margin-top: 18px; margin-left: 350px; background-image: none; }
#red .ui-slider-range { background: #ef2929; }
#red .ui-slider-handle { border-color: #ef2929; }
#green .ui-slider-range { background: #8ae234; }
#green .ui-slider-handle { border-color: #8ae234; }
#blue .ui-slider-range { background: #729fcf; }
#blue .ui-slider-handle { border-color: #729fcf; }
#demo-frame > div.demo { padding: 10px !important; }
</style>
<title>上机练习</title>
</head>
<body>
<div id="red"></div>
<div id="green"></div>
<div id="blue"></div>
<div id="swatch" class="ui-widget-content ui-corner-all"></div>
</body>
</html>

```

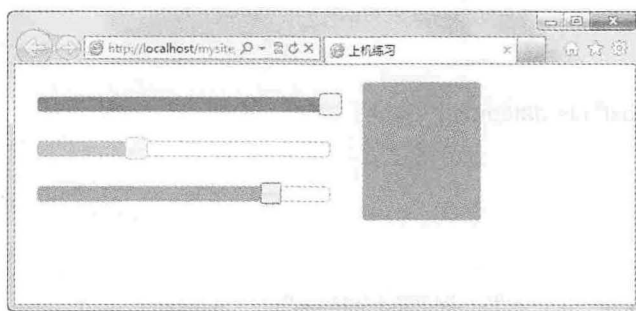


图 14.14 设计调色板

14.5 日期选择器

日期选择器组件是 jQuery UI 库中最为精致且支持文档最为丰富的组件。日期选择器组件为站点或者应用的访客们提供了非常简便的选择日期的界面。无论表单中哪个域需要输入日期，都可以为之添加一个日

期选择器组件，它允许访问者使用具有吸引力的美观的组件进行输入，而与此同时后台可以根据需要获取特定格式的日期。

日期选择器组件内建的功能包括自动打开和关闭动画，以及使用键盘操作组件的界面。当按下 **Ctrl** 键时，使用键盘上的箭头就能够选择一个日期单元格，然后使用 **Enter** 键就可以选中该日期。虽然日期选择器组件易于创建和配置，但是它是一个复杂的组件，由一系列基本元素构成。尽管日期选择器是复杂的，但是与已经学过的其他 UI 库的组件一样，可以只用一行代码实现默认的日期选择器。

添加日期选择器部件需要在页面中引入下面几个 JavaScript 文件，然后为目标日期选择器包含框绑定 `datepicker()` 构造函数即可。

- ☒ jquery-1.3.js +
- ☒ jquery.ui.core.js
- ☒ jquery.ui.datepicker.js
- ☒ jquery.ui.all.css

【示例 5】 在网页中为 `<input type="text" id="datepicker">` 文本框绑定一个日期选择器。演示效果如图 14.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.datepicker.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#datepicker" ).datepicker();
});
</script>
<style type="text/css">
</style>
<title>上机练习</title>
</head>
<body>
<p>Date: <input type="text" id="datepicker"></p>
</body>
</html>
```



图 14.15 基本日期选择器效果

基本日期选择器使用比较简单，直接在对应的文本框上调用 `datepicker()` 函数即可。当然日期选择器组件具有大量可以配置的属性，使用这些属性，用户能够精确控制日期选择器功能和外观，说明如表 14.11 所示。

表 14.11 日期选择器组件的属性

属 性	说 明
<code>altField</code>	为日期选择器指定可选的 <code>input</code> 域，所选中的日期将会被添加到该域中。默认值为空
<code>altFormat</code>	添加到 <code>input</code> 域的日期指定一个可选项的格式。默认值为空
<code>appendText</code>	在日期选择器 <code>input</code> 域的后面附加文本，以显示所选日期的格式
<code>buttonImage</code>	设置触发按钮所用的图片路径
<code>buttonImageOnly</code>	如果设置 <code>true</code> ，则会使用图片代替触发按钮
<code>buttonText</code>	触发按钮上显示的文本。默认值为...
<code>changeFirstDay</code>	当一个日期标题被单击时，重新排列日历。默认值为 <code>true</code>
<code>changeMonth</code>	下拉框显示月历变化。默认值为 <code>true</code>
<code>changeYear</code>	下拉框显示年度变化。默认值为 <code>true</code>
<code>closeAltTop</code>	在日历顶部显示关闭按钮。默认值为 <code>true</code>
<code>constrainInput</code>	限制文本框输入的必须是日期格式。默认值为 <code>true</code>
<code>defaultDate</code>	设置初始时选择器高亮显示的日期。默认值为 <code>null</code>
<code>duration</code>	设置日期选择器打开的速度。默认值为 <code>normal</code>
<code>goToCurrent</code>	将当前日期链接设置为日期选择器当前选中的日期，而不是本日日期。默认值为 <code>false</code>
<code>hideIfNoPrevNext</code>	如果没有上一页或下一页，则隐藏 <code>Pre/Next</code> 链接。默认值为 <code>false</code>
<code>isRTL</code>	将日期格式设置为从右到左。默认值为 <code>false</code>
<code>mandatory</code>	强制必须显示日期。默认值为 <code>false</code>
<code>maxDate</code>	设置能够选择的最大日期。默认值为 <code>null</code>
<code>minDate</code>	设置能够选择的最小日期。默认值为 <code>null</code>
<code>navigationAsDateFormat</code>	允许将月份名称作为 <code>Prev</code> 、 <code>Next</code> 和 <code>Current</code> 链接名称。默认值为 <code>false</code>
<code>numbersOfMonths</code>	设置单个日期选择器所显示的月份数。默认值为 1
<code>rangeSeparator</code>	在使用范围时设置两个日期期间的分隔符。默认值为 <code>-</code>
<code>rangeSelect</code>	激活日期范围的选择。默认值为 <code>false</code>
<code>shortYearCutoff</code>	当使用两个数字表示月份时，该属性用于确定当前所在的地址。默认值为 <code>+10</code>
<code>showOn</code>	设置显示日期选择器的事件。默认值为 <code>focus</code>
<code>showOtherMonths</code>	显示上个月的第一天和下个月的第 1 天。默认值为 <code>false</code>
<code>showStatus</code>	在日期选择器中显示状态条。默认值为 <code>false</code>
<code>showWeeks</code>	显示星期数，即列数。默认值为 <code>false</code>
<code>showAnim</code>	设置日期选择器打开和关闭时执行的动画。默认值为 <code>show</code>
<code>showOptions</code>	设置额外的动画配置选项。默认值为 <code>{}</code>
<code>stepMonths</code>	设置 <code>prev</code> 和 <code>next</code> 链接所前进或后退的月数。默认值为 1
<code>yearRange</code>	在年度下拉列表框中指定可选年份的范围。默认值为 <code>-10+10</code>

除了上面介绍的属性外，日期选择器组件还提供了很多事件，专门用于特定事件执行代码的回调函数，说明如表 14.12 所示。

表 14.12 日期选择器组件事件

事 件	说 明
<code>beforeShow</code>	接收一个用于定制日期选择器的配置对象
<code>beforeShowDay</code>	用于预选择特定的日期

事 件	说 明
calculateWeek	改变计算一年中第几个星期的计算方法
onSelect	设置选择事件的回调函数
onChangeMonthYear	设置当前月份或者年度改变时要执行的回调函数
onClose	设置关闭事件的回调函数
statusForDate	确定状态条文本是否正显示信息的函数

日期选择器是直观而易用的，但是如果获得超出之前所展示的效果，那么就需要它的内建方法，如表 14.13 所示。

表 14.13 日期选择器组件的方法

方 法	说 明
change	使用配置对象改变已经存在的日期选择器
destroy	断开并删除所有关联的日期选择器
dialog	在对话框组件中打开日期选择器
disable	禁用一个文本框，并因此关联日期选择器
enable	激活一个被禁用的文本框，连同相应的日期选择器
getDate	获取当前选择的日期
hide	以编程方式关闭日期选择器
isDisabled	确定日期选择器是否被禁用
setDate	以编程方式选择日期
show	以编程方式显示日期选择器

1. 在日期选择器中显示按钮

下面示例演示了如何在日期选择器面板中显示控制按钮。演示效果如图 14.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.datepicker.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $( "#datepicker" ).datepicker({
        showButtonPanel: true
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<p>Date: <input type="text" id="datepicker"></p>
```

```
</body>
</html>
```



图 14.16 在日期选择器中显示按钮

2. 在日期选择器中显示年月下拉菜单

下面示例演示了如何在日期选择器中显示年月下拉菜单。演示效果如图 14.17 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.datepicker.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("#datepicker").datepicker({
        changeMonth: true,
        changeYear: true
    });
});
</script>
<title>上机练习</title>
</head>
<body>
<p>Date: <input type="text" id="datepicker"></p>
</body>
</html>
```

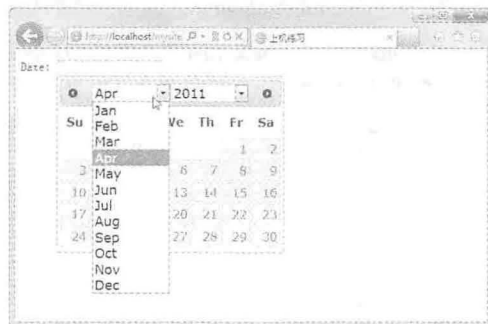


图 14.17 在日期选择器中显示年月下拉菜单

3. 设置日期范围

下面示例演示了如何设置日期范围，当设置起始日期之后，则结束日期必须在起始日期之后。同时本案例还演示了如何设置连续显示 3 个月份的月历。演示效果如图 14.18 所示。


```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.ui.datepicker.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
    $(function() {
        var dates = $( "#from, #to" ).datepicker({
            defaultDate: "+1w",
            changeMonth: true,
            numberOfMonths: 3,
            onSelect: function( selectedDate ) {
                var option = this.id == "from" ? "minDate" : "maxDate",
                    instance = $( this ).data( "datepicker" ),
                    date = $.datepicker.parseDate(
                        instance.settings.dateFormat ||
                        $.datepicker._defaults.dateFormat,
                        selectedDate, instance.settings );
                dates.not( this ).datepicker( "option", option, date );
            }
        });
    });
</script>
<title>上机练习</title>
</head>
<body>
<label for="from">从</label>
<input type="text" id="from" name="from"/>
<label for="to">到</label>
<input type="text" id="to" name="to"/>
</body>
</html>
```



图 14.18 设置日期范围

第15章

jQuery UI 特效开发

( 视频讲解：49 分钟)

jQuery UI 特效是独立于 jQuery UI 组件而单独开发的动画库，与其他独立组件一样，特效也需要一个单独的核心库文件提供服务，包括封装元素、控制动画等，但同时大多数特性还应有自己的源文件。在 jQuery UI 特效库中提供了众多动画特效，说明如图 15.1 所示。

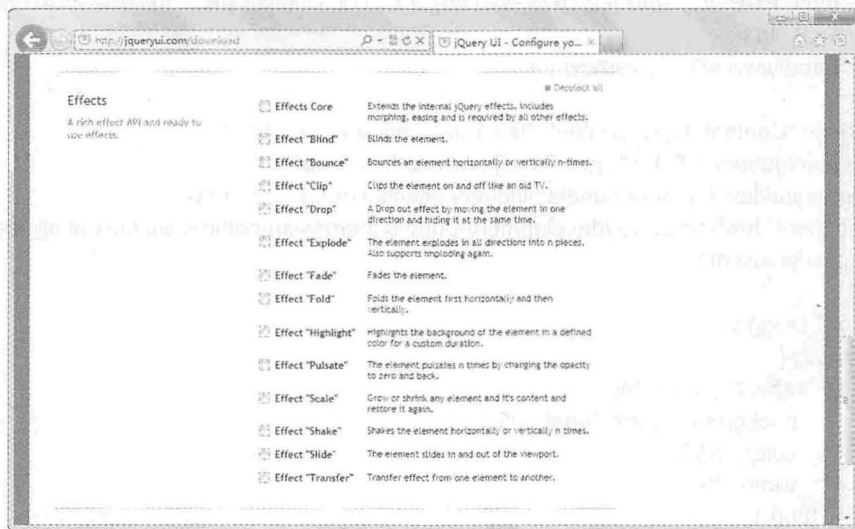


图 15.1 jQuery UI Effects

使用特效时只需要在页面上包含核心文件（jquery.effects.core.js）和该特效本身的源文件，不过与 jquery.ui.core.js 文件不同，jquery.effects.core.js 被设计为部分而非完全独立的。当单独使用核心特效文件时，可以利用它提供的颜色改变动画，如平滑地将元素的背景色转换成另一种颜色、样式类切换和更高级的缓存动画等。jQuery UI 特效库包含如下几种特效：

- ☒ Blind（百叶窗）
- ☒ Bounce（弹跳）
- ☒ Clip（剪辑）
- ☒ Drop（落体）
- ☒ Explode（爆炸）
- ☒ Fade（渐隐）

- ☑ Fold (折叠)
- ☑ Highlight (高亮)
- ☑ Pulsate (抖动)
- ☑ Scale (缩放)
- ☑ Shake (摇晃)
- ☑ Slide (滑动)
- ☑ Transfer (转换)

15.1 特效核心

jquery.effects.core.js 文件是 jQuery UI 特效库的核心，它可以单独使用，实际上就是在通用 jQuery 库基础上。当然如果不需要其他特效，它可以不与特定的特效文件搭配使用。

animate()方法是 jQuery 动画基础，它是属于 jQuery 库的，不属于 jQuery UI 库。但是 jquery.effects.core.js 文件扩展了 animate()方法，以用来处理颜色和样式类等相关的动画。

【示例 1】 演示当单击按钮后，会使用 animate()方法将一系列新的 CSS 属性应用到目标元素上，这样样式属性由一个常量对象以属性名值对的形式提供。演示效果如图 15.2 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#button" ).toggle(
        function() {
            $( "#effect" ).animate({
                backgroundColor: "#aa0000",
                color: "#fff",
                width: 500
            }, 1000 );
        },
        function() {
            $( "#effect" ).animate({
                backgroundColor: "#fff",
                color: "#000",
                width: 240
            }, 1000 );
        }
    );
});
</script>
<style type="text/css">
.toggle { width: 500px; height: 200px; position: relative; }
#button { padding: .5em 1em; text-decoration: none; }
```

```

#effect { width: 240px; height: 135px; padding: 0.4em; position: relative; background: #fff; }
#effect h3 { margin: 0; padding: 0.4em; text-align: center; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
  <div id="effect" class="ui-widget-content ui-corner-all">
    <h3 class="ui-widget-header ui-corner-all">动画演示</h3>
    <p>颜色动画综合演示</p>
  </div>
</div>
<a href="#" id="button" class="ui-state-default ui-corner-all">动画切换</a>
</body>
</html>

```



图 15.2 颜色特效效果

颜色动画可供使用的 CSS 样式属性包括以下几种：

- ☑ backgroundColor (背景色)
- ☑ borderColor (边框色, 包括任意边)
- ☑ color (前景色)
- ☑ outlineColor (轮廓色)

颜色可以使用 RGB 或者 HEX 格式指定, 甚至可以使用标准的颜色名称。

除了颜色外, jquery.effects.core.js 核心文件提供了整个样式类变换的动画, 可用于平滑、准确地切换样式, 而不是以唐突、剧烈的变化方式。

【示例 2】 本示例演示了当单击按钮后, 方形盒子会自动平滑地缩放。演示效果如图 15.3 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
  $( "#button" ).click(function() {
    $( "#effect" ).toggleClass( "newClass", 1000 );
    return false;
  });
});

```



```

    });
  });
</script>
<style type="text/css">
.toggler { width: 500px; height: 200px; position: relative; }
#button { padding: .5em 1em; text-decoration: none; }
#effect { position: relative; width: 240px; padding: 1em; letter-spacing: 0; font-size: 1.2em; border: 1px solid #000;
background: #eee; color: #333; }
.newClass { text-indent: 40px; letter-spacing: .4em; width: 410px; height: 100px; padding: 30px; margin: 10px;
font-size: 1.6em; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
  <div id="effect" class="newClass ui-corner-all">尺寸缩放动画演示</div>
</div>
<a href="#" id="button" class="ui-state-default ui-corner-all">动画演示</a>
</body>
</html>

```



图 15.3 样式动画效果

从效果上看,页面的功能和上一示例相同,但本示例使用的是以样式类切换的方式实现,它同样可以包含颜色样式规则。当然,还可以添加样式、移出样式、样式切换等动画。

标准的 jQuery 的 `animate()` 方法具有一些内建的基本缓冲功能,但是对于更高级的缓冲效果,则必须包含额外的缓冲插件。`jquery.effects.core.js` 文件具有所有高级缓冲选项,因此不需要再包含额外的插件,就可以完成各种复杂的任务。

15.2 高 亮

高亮特效指的是任何调用该方法的元素都被设置为高亮显示效果。添加该特效需要在页面中引入下面几个 JavaScript 文件:

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.highlight.js

【示例 3】演示当单击按钮之后,面板背景色会高亮显示一下。演示效果如图 15.4 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.highlight.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#button" ).click(function(){
        $( "#effect" ).effect("highlight");
    })
});
</script>
<style type="text/css">
#button { padding: .5em 1em; text-decoration: none; }
#effect { width: 240px; height: 135px; padding: 0.4em; margin: 1em; }
#effect h3 { margin: 0; padding: 0.4em; text-align: center; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
    <div id="effect" class="ui-widget-content ui-corner-all">
        <h3 class="ui-widget-header ui-corner-all">特效演示</h3>
        <p>高亮效果</p>
    </div>
</div>
<button id="button" class="ui-state-default ui-corner-all">特效</button>
</body>
</html>

```



图 15.4 高亮效果

在示例 3 中高亮效果可能不是很明显，但是在高端界面中 `highlight` 特效适合作为帮助工具。如果有一系列动作需要以特定的顺序完成，那么高亮效果可以即时提醒访问者接下来要完成的步骤，同样它也可以用于电子版的教程或手册，以提示屏幕上的特定内容。

`effect()` 构造函数是 jQuery UI 特效的唯一接口，它不仅接收表示使用何种特效的字符串参数，还可以接收 3 个附加的参数以控制特效的功能，这些参数是可选的，说明如下：

- ☑ 一个包含附件配置属性的对象。

- ☑ 一个整数，代表了特效所持续的毫秒数，或者一个字符串，如 slow、normal、fast。
- ☑ 一个回调函数，在特效结束时执行。

【示例 4】针对示例 3，重新设计高亮特效，让高亮效果显示 2 秒钟，同时在显示完毕后，修改段落文本内容，以便进行提示。演示效果如图 15.5 所示。

```
$(function() {
    $("#button").click(function(){
        $("#effect").effect("highlight",{},2000,function(){
            $(this).children("p").text("高亮效果已经完成");
        });
    })
});
```



图 15.5 缓慢的高亮效果

在上面代码中，使用一个空对象作为 effect() 构造函数的第 2 个参数，这是因为本示例不需要任何额外的配置信息，但是仍然需要提供一个空对象参数，以便传递第 3 个和第 4 个参数。

用于作为高亮特效的配置对象只包含 2 个配置属性，说明如表 15.1 所示。

表 15.1 高亮配置参数

属 性	说 明
color	设置高亮颜色。默认值为亮黄色（#ffff99）
mode	设置高亮模式，包括 show 和 hide 两个值。默认值为 show

【示例 5】针对示例 4，重新设置配置参数。设置高亮色为红色，同时在高亮动画之后，隐藏高亮面板。

```
$(function() {
    $("#button").click(function(){
        $("#effect").effect("highlight",{ color:"red",mode:"hide"},2000,function(){
            $(this).children("p").text("高亮效果已经完成");
        });
    })
});
```

15.3 弹 跳

弹跳特效指的是任何调用该方法的元素都被设置为弹跳显示效果。添加该特效需要在页面中引入下面几个 JavaScript 文件：

- ☑ jquery-1.3.js +
- ☑ jquery.effects.core.js

☑ jquery.effects.bounce.js

【示例 6】演示当单击按钮之后，面板会上下弹跳几下，然后恢复默认显示状态。演示效果如图 15.6 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.bounce.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#button" ).click(function(){
        $( "#effect" ).effect("bounce");
    }
});
</script>
<style type="text/css">
#button { padding: .5em 1em; text-decoration: none; }
#effect { width: 240px; height: 135px; padding: 0.4em; margin: 1em; }
#effect h3 { margin: 0; padding: 0.4em; text-align: center; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
    <div id="effect" class="ui-widget-content ui-corner-all">
        <h3 class="ui-widget-header ui-corner-all">特效演示</h3>
        <p>弹跳效果</p>
    </div>
</div>
<button id="button" class="ui-state-default ui-corner-all">弹跳</button>
</body>
</html>
```



图 15.6 弹跳效果

用于作为弹跳特效的配置对象包含 4 个配置属性，说明如表 15.2 所示。

表 15.2 弹跳配置参数

属 性	说 明
direction	设置弹跳方向, 包括 up、down、left、right。默认值为 up
distance	设置和弹跳的距离。默认值为 20 像素
mode	设置弹跳模式, 包括 show、hide、effect 3 个值。默认值为 effect
times	设置弹跳次数。默认值为 5

【示例 7】 针对示例 6, 重新设置配置参数。设置弹跳方向为向下, 弹跳距离为 10 像素, 模式为弹跳完成之后显示, 弹跳次数为两次, 同时弹跳动画持续时间为 20 毫秒, 弹跳完成之后修改面板包含的提示后文本。演示效果如图 15.7 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.bounce.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("#button").click(function(){
        $("#effect").effect("bounce", {
            direction: "down",
            distance: 10,
            mode: "show",
            times: 2
        },200,function(){
            $(this).children("p").text("弹跳效果已经完成");
        });
    });
});
</script>
<style type="text/css">
#button { padding: .5em 1em; text-decoration: none; }
#effect { width: 240px; height: 135px; padding: 0.4em; margin: 1em; }
#effect h3 { margin: 0; padding: 0.4em; text-align: center; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
    <div id="effect" class="ui-widget-content ui-corner-all">
        <h3 class="ui-widget-header ui-corner-all">特效演示</h3>
        <p>弹跳效果</p>
    </div>
</div>
<button id="button" class="ui-state-default ui-corner-all">弹跳</button>
</body>
</html>
```

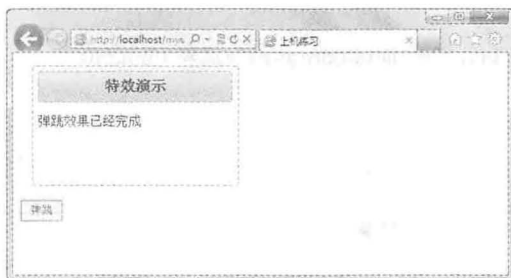


图 15.7 自定义弹跳效果

15.4 摇 晃

摇晃特效与弹跳特效类似，指的是任何调用该方法的元素都被设置为摇晃显示效果。添加该特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.shake.js

【示例 8】演示当单击按钮之后，面板会摇晃几下，然后恢复默认状态。演示效果如图 15.8 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.shake.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $( "#button" ).click(function(){
        $( "#effect" ).effect("shake ");
    }
});
</script>
<style type="text/css">
#button { padding: .5em 1em; text-decoration: none; }
#effect { width: 240px; height: 135px; padding: 0.4em; margin: 1em; }
#effect h3 { margin: 0; padding: 0.4em; text-align: center; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="toggler">
    <div id="effect" class="ui-widget-content ui-corner-all">
        <h3 class="ui-widget-header ui-corner-all">特效演示</h3>
        <p>摇晃效果</p>
    </div>
```

```

</div>
<button id="button" class="ui-state-default ui-corner-all">摇晃</button>
</body>
</html>

```



图 15.8 摇晃效果

用于作为摇晃特效的配置对象包含 3 个配置属性, 说明如表 15.3 所示。

表 15.3 摇晃配置参数

属 性	说 明
direction	设置摇晃方向, 包括 up、down、left、right 4 个值。默认值为 left
distance	设置和摇晃的距离。默认值为 20 像素
times	设置摇晃次数。默认值为 3

15.5 转 换

转换特效与其他特效不同, 它并非直接影响目标元素, 而是将特定元素的轮廓传送到另一个指定的元素中。添加该特效需要在页面中引入下面几个 JavaScript 文件:

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.transfer.js

【示例 9】演示当单击 div 盒子时, 会自动把自己转换为另一个盒子的轮廓, 并设置动画持续时间为 1000 毫秒。演示效果如图 15.9 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.transfer.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("div").click(function () {
        var i = 1 - $("div").index(this);
        $(this).effect("transfer", {
            to: $("div").eq(i)

```

```

    }, 1000);
  });
</script>
<style type="text/css">
div.green { margin: 0px; width: 100px; height: 80px; background: green; border: 1px solid black; position:
relative; }
div.red { margin-top: 10px; width: 50px; height: 30px; background: red; border: 1px solid black; position:
relative; }
.ui-effects-transfer { border: 2px solid black; }
</style>
<title>上机练习</title>
</head>
<body>
<div class="green"></div>
<div class="red"></div>
</body>
</html>

```



图 15.9 转换效果

用于作为转换特效的配置对象包含两个配置属性，说明如表 15.4 所示。

表 15.4 转换配置参数

属 性	说 明
className	为代表传送对象的元素应用的类样式
to	设置要转换的匹配元素

15.6 缩 放

缩放特效用于缩小或者放大指定的元素。添加该特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.scale.js

【示例 10】 演示当单击 div 盒子时，该盒子会自动水平放大一倍宽度，动画演示持续时间为 1000 毫秒。演示效果如图 15.10 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

```



```

<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.scale.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("div").click(function () {
        $(this).effect("scale", {
            percent: 200,
            direction: 'horizontal'
        }, 1000);
    });
});
</script>
<style type="text/css">
div { margin: 0px; width: 100px; height: 80px; background: green; border: 1px solid black; position: relative; }
</style>
<title>上机练习</title>
</head>
<body>
<div></div>
</body>
</html>

```

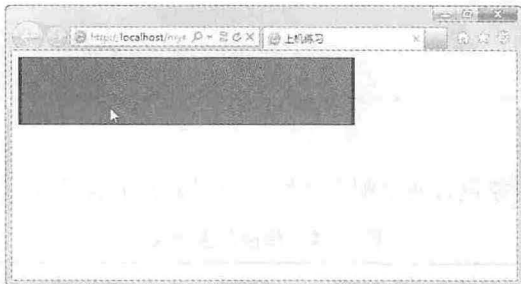


图 15.10 缩放效果

用于作为缩放特效的配置对象包含两个配置属性，说明如表 15.5 所示。

表 15.5 缩放配置参数

属 性	说 明
direction	设置缩放的方向，包括 both、vertical、horizontal 3 个值。默认值为 both，即高和宽同时缩放
from	设置缩放元素的开始大小，对象类型，如 { height: ..., width: .. }。默认为空对象
origin	设置消失点，数组类型，可用于 show/hide 动画。默认值为 ['middle','center']
percent	设置缩放元素结束时的尺寸。默认值为 0，百分比单位
scale	设置缩放的区域，包括 both、box、content 3 个值。默认值为 both

15.7 爆 炸

爆炸特效是一种真正令人惊叹的特效，它可以使目标元素在彻底消失前分解为指定数目的切片。该特

效易于使用,只需要极少数的代码以及极少数的配置属性。添加该特效需要在页面中引入下面几个 JavaScript 文件:

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.explode.js

【示例 11】 演示当单击网页中的图片时,图片会以爆炸形式在 1000 毫秒内快速消失。演示效果如图 15.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.explode.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("img").click(function () {
        $(this).hide("explode", 1000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>
```



图 15.11 爆炸效果

用于作为爆炸特效的配置对象包含两个配置属性,说明如表 15.6 所示。

表 15.6 爆炸配置参数

属 性	说 明
mode	设置爆炸的模式,包括 show 和 hide 两个值。默认值为 hide
pieces	设置爆炸时产生的碎片数。默认值为 9

【示例 12】 演示当单击网页中的图片时,图片会以爆炸形式,在 1000 毫秒内快速被炸飞为 16 片,然后慢慢合并在一起。演示效果如图 15.12 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.explode.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("img").click(function () {
        $(this).hide("explode", 1000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>

```

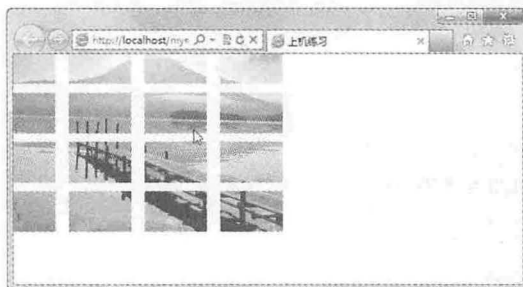


图 15.12 自定义爆炸效果

15.8 抖 动

抖动特效是一种与元素透明度相关的特效，它能够按指定的次数短暂降低元素透明度，使其看起来如同脉动效果一般。添加该特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.pulsate.js

【示例 13】演示当单击网页中的图片时，图片会以抖动形式在 2000 毫秒内快速闪现 3 次。演示效果如图 15.13 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />

```

```
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.pulsate.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("img").click(function () {
        $(this).effect("pulsate", {
            times:3
        }, 2000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>
```

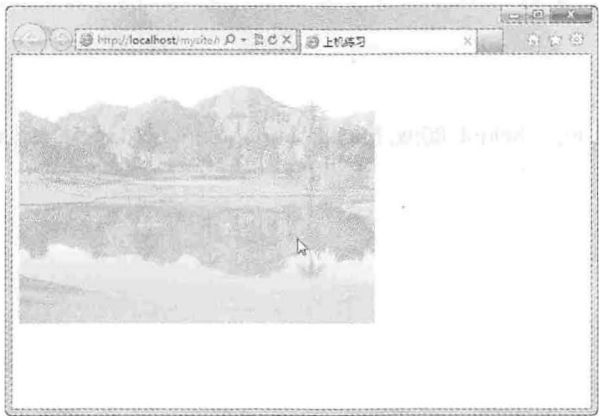


图 15.13 抖动效果

用于作为抖动特效的配置对象包含两个配置属性，说明如表 15.7 所示。

表 15.7 抖动配置参数

属 性	说 明
mode	设置抖动的模式，包括 show 和 hide 两个值。默认值为 show
times	设置抖动次数。默认值为 5

15.9 落 体

落体特效非常简单，它帮助元素实现从页面中下落的效果，该特效同时会调整元素的高度和透明度。落体特效应用的场合比较多，如提示信息、弹出对话框等。添加该特效需要在页面中引入下面几个 JavaScript 文件：

- ☑ jquery-1.3.js +
- ☑ jquery.effects.core.js
- ☑ jquery.effects.drop.js

【示例 14】 演示当单击网页中的 div 盒子，它就会自动下落，并逐渐渐隐消失。演示效果如图 15.14 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.drop.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("div").click(function () {
        $(this).hide("drop", { direction: "down" }, 1000);
    });
});
</script>
<style type="text/css">
div { margin: 0px; width: 100px; height: 80px; background: green; border: 1px solid black; position: relative; }
</style>
<title>上机练习</title>
</head>
<body>
<div></div>
</body>
</html>
```

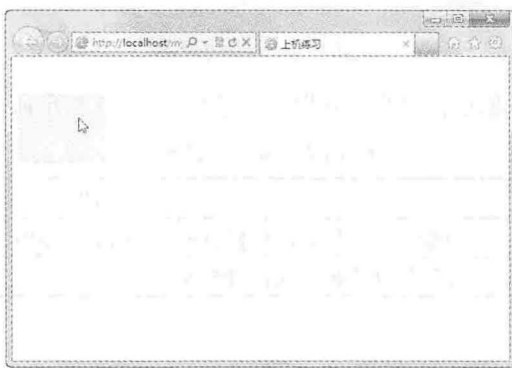


图 15.14 落体效果

【提示】

通过使用 hide()、show()等方法，在调用这些方式时恰当设置动画方式，即 jQuery UI 特效类型字符串，也可以使用 effect()构造函数来设计。例如针对上面示例也可以这样调用，代码如下（后面几种特效用法类似，就不再一一说明）：

```
$(function() {
    $("div").click(function () {
        $(this).effect("drop", {
            direction: "down",
            mode: "hide",
        }, 1000);
    });
});
```

用于作为落体特效的配置对象包含两个配置属性，说明如表 15.8 所示。

表 15.8 落体配置参数

属 性	说 明
mode	设置落体的模式，包括 show 和 hide 两个值。默认值为 hide
direction	设置落体方向，包括 left、right、up、down 共 4 个值。默认值为 left

15.10 滑 动

上面介绍的几种特效主要控制元素的透明度，下面几种特效不再控制元素的透明度，而是以各种方式显示或者隐藏元素。滑动特效使元素在显示或者隐藏时具有向页面中或者页面外滑行的效果，这与落体特效很相似，但是它们的主要区别在于并不利用元素的透明度。添加滑动特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.slide.js

【示例 15】 演示当单击网页中图片时，它会自动向下滑动并消失。演示效果如图 15.15 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.slide.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("img").click(function () {
        $(this).hide("slide", { direction: "down" }, 2000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>
```

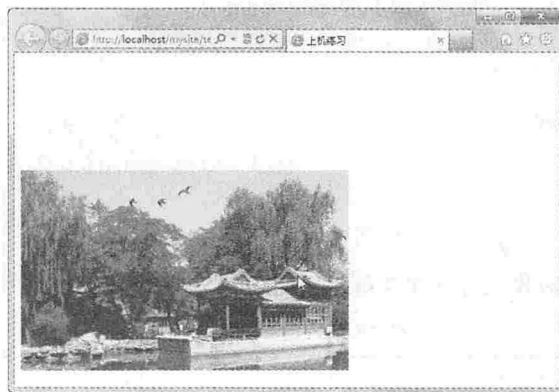


图 15.15 滑动效果

用于作为滑动特效的配置对象包含两个配置属性，说明如表 15.9 所示。

表 15.9 滑动配置参数

属 性	说 明
mode	设置滑动的模式，包括 show 和 hide 两个值。默认值为 show
direction	设置滑动方向，包括 left、right、up、down 4 个值。默认值为 left
distance	设置滑动距离，默认值为 el.outerWidth。可设置小于元素的宽度或者高度的任意整数

15.11 剪 辑

剪辑特效与滑动特效很相似，主要不同之处在于，滑动特效是通过使目标元素的一边向另一边移动来获取滑动效果，而剪辑特效则通过将目标元素的两边向中心移动实现。添加剪辑特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.clip.js

【示例 16】 演示当单击网页中图片时，它会自动从两侧向中间进行剪辑，最后消失。演示效果如图 15.16 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.clip.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("img").click(function () {
        $(this).hide("clip", { direction: "horizontal" }, 1000);
    });
});
```

```

</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>

```

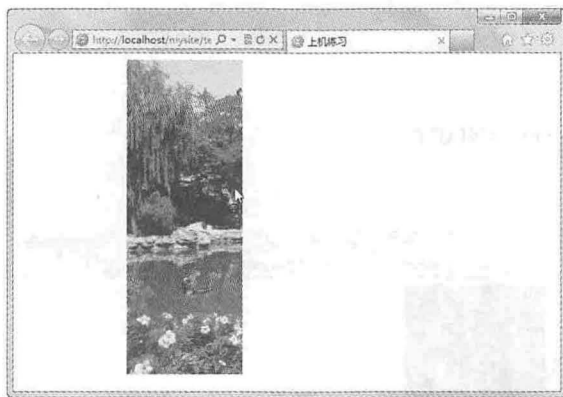


图 15.16 剪辑效果

用于作为剪辑特效的配置对象包含两个配置属性，说明如表 15.10 所示。

表 15.10 剪辑配置参数

属 性	说 明
mode	设置剪辑的模式，包括 show 和 hide。默认值为 hide
direction	设置剪辑方向，包括 vertical 和 horizontal 两个值。默认值为 vertical

15.12 百 叶 窗

百叶窗特效与滑动特效很相似。从视觉效果上看，两种特效的目标元素的行为都是相同的，并且它们的代码文件也非常相似。这两种特效间的主要区别在于百叶窗特效只能够指定特效的方向轴，而不是实际的方向。添加百叶窗特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.blind.js

【示例 17】 演示当单击网页中图片时，它会自动向左侧收缩，最后消失。演示效果如图 15.17 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.blind.js"></script>

```



```

<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript">
$(function() {
    $("img").click(function () {
        $(this).hide("blind", { direction: "horizontal" }, 1000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>

```

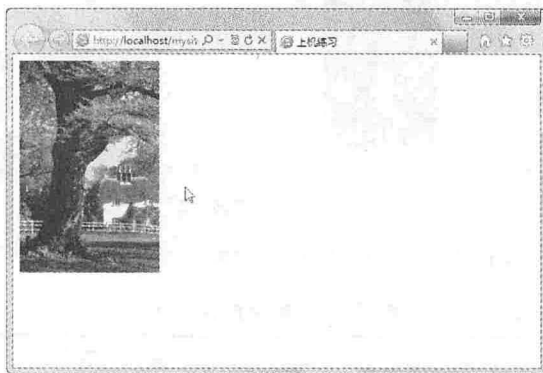


图 15.17 百叶窗效果

用于作为百叶窗特效的配置对象包含两个配置属性，说明如表 15.11 所示。

表 15.11 百叶窗配置参数

属 性	说 明
mode	设置百叶窗的模式，包括 show 和 hide。默认值为 hide
direction	设置百叶窗方向，包括 vertical 和 horizontal 两个值。默认值为 vertical

15.13 折 叠

折叠是一种简洁的特效，它能够使元素如一张纸一样被折叠起来，其具体实现方式是将指定的元素底部移到距离顶边 15 个像素的位置，然后再将元素右边线完全移动到左边线的位置。在该特效的第 1 个阶段，元素被缩小到距离顶边距离可以由 API 所开放的配置属性设置，该属性接收一个整型值，在实际应用中可以将该距离设置为合适的大型。添加特效需要在页面中引入下面几个 JavaScript 文件：

- ☒ jquery-1.3.js +
- ☒ jquery.effects.core.js
- ☒ jquery.effects.fold.js

【示例 18】演示当单击网页中图片时，它会自动向上和向左折叠图片，最后消失。演示效果如图 15.18 所示。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<script src="jQuery/jquery-1.6.4.js" type="text/javascript"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.core.js"></script>
<script src="jqueryui/development-bundle/ui/jquery.effects.fold.js"></script>
<link rel="stylesheet" href="jqueryui/development-bundle/themes/smoothness/jquery.ui.all.css">
<script type="text/javascript" >
$(function() {
    $("img").click(function () {
        $(this).hide("fold", {}, 1000);
    });
});
</script>
<title>上机练习</title>
</head>
<body>

</body>
</html>

```

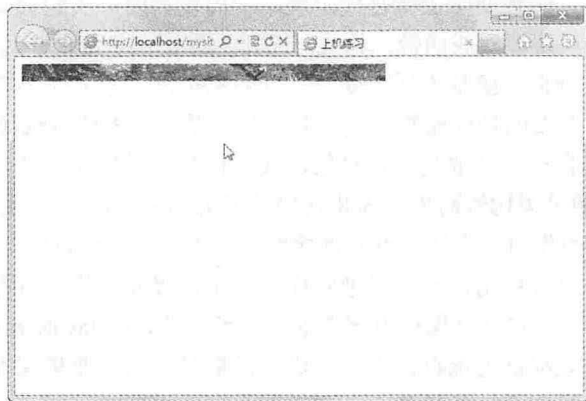


图 15.18 折叠效果

用于作为折叠特效的配置对象包含 3 个配置属性，说明如表 15.12 所示。

表 15.12 折叠配置参数

属 性	说 明
horizFirst	设置是否在水平方向先折叠。默认值为 false
mode	设置动画模式，包括 show 和 hide 两个值。默认值为 hide
size	设置元素被折叠的尺寸，整型。默认值为 15

第16章

jQuery 框架透析之函数式基础

( 视频讲解：2 小时 40 分钟)

函数是一种映射关系，它构成了 JavaScript 编程基础。JavaScript 虽然是一门基于对象的语言，但是函数却是 JavaScript 第一型（First-class Data Types，即第一类数据类型），它构成了 JavaScript 对象类型基础，对象（Object）却是以函数（Function）为基类实现的。当然 JavaScript 又是一门函数式编程语言。

在 JavaScript 语言中，函数可以有下面多种理解：

- ☑ 从程序的角度来分析，函数实际上就是代码块，一段被封闭严实的代码块。在程序设计中，常把一些重复使用的功能模块编写成函数，构成一个函数库，以供随时调用。
- ☑ 从数据结构的角度来分析，函数本质上就是一种数据类型，它就是具有复杂结构的数据体。函数不仅是用来避免重复开发的功能模块，更重要的是它是一个能够装载数据的容器。
- ☑ 从面向对象的角度来分析，函数是一种对象，或者说是一类抽象对象，即所谓的类（或称为构造函数），所有对象都通过类构造而来，因此它构成了 JavaScript 对象的原型。
- ☑ 在函数式语言中，函数作为一个运算单元直接参与到表达式的运算中。在 JavaScript 中，不仅可以把函数作为一个对象进行引用，还可以把它作为一个运算元（值）进行计算。在函数式编程中，函数由于自身的封闭性，它具有保存内部数据的先天条件。在 JavaScript 函数中，函数内的私有变量可以被修改，而且能够延迟保存。当下次进入函数时，局部变量保存的数据依然存在。作为闭包，函数具有天生的封闭性，外界是不允许访问内部数据的。同时，闭包是无副作用的，它不会影响外部环境的运行。
- ☑ 在表达式运算中，求值是运算的核心。函数作为表达式中的一个运算元，它也具有值的含义。不管函数内部是多么复杂，最终返回的只是一个值，而不是一个复杂的结构或引用。由于函数的封装性，使它具有值运算的能力，且不会影响外部的环境。因此，应学会使用函数来封装表达式和语句，并把函数作为运算元参与表达式的运算。
- ☑ 从语义上分析，函数的调用过程实际上就是表达式运算中求值的过程。从这一点来看，在函数式编程中，函数是一种高效的连续运算的工具。如对于循环结构来说，使用函数递归运算会存在很大的系统损耗，但是如果把循环语句封装在函数结构中，然后把函数作为值参与表达式的运算，实际上也是高效实现循环结构表达式化。

总之，函数式编程中的函数不是结构、过程，也不是代码封装块，它们都是命令式语言中函数的基本特质。函数其实就是运算的符号，作为运算元参与表达式的计算。实际上，可以把函数看做是一个运算符（从逻辑上分析），函数的参数是该运算符的运算元，返回值是运算的结果。由于函数固有的自闭性，其表现为对内严密封装，对外不干扰别人，没有副作用，可以说它是最安全的一个复杂运算符了。从外部来看，

函数封装语句，这样利用函数这个特殊的运算符，使连续运算成为可能，同时利用函数自身的优势来组织代码。

16.1 定义函数

定义函数的方法有多种（如下所示），不同的方法体现了不同的设计理念，不同的定义方法适合在不同的语境中。

☑ function 语句

//使用 function 语句编织函数

```
function f(x){
    return x;
}
```

☑ Function()构造函数

//使用 Function()构造函数克隆函数

```
var f = new Function("x", "return x;");
```

☑ 函数直接量

//使用函数直接量直接生成函数

```
var f = function(x){
    return x;
}
```

不管使用哪种方法定义函数，它们都是 Function 对象的实例，或者说都是 Function 对象的后代，并将继承 Function 对象所有默认或自定义的方法和属性，可以把 Function 对象视为函数模板和元类。

16.1.1 构造函数

构造函数也是一类函数，本质上分析它是函数结构的模型，是类（面向对象的基础）的基本构成要素和成员。

在默认状态下，JavaScript 预定义了很多构造函数，如 Function()、Array()、Date()、string()等。如果去掉小括号，它们实际上就是 JavaScript 的内部对象。当然也可以定义一个属于自己的构造函数。例如，定义一个自定义构造函数 Me()，代码如下：

```
function Me(){           //自定义构造函数结构
    this.name = "张三";   //自定义构造函数的属性
    this.address = "中国上海"; //自定义构造函数的属性
    this.saying = function(){ //自定义构造函数的方法
        return "Hello, world";
    }
}
```

一般构造函数名的首字母要大写，以便与普通函数相区别，并通过点号运算符指定构造函数的成员（如属性和方法）。针对上面自定义构造函数，可以使用如下方法实例化并调用对象：

```
var a = new Me();           //实例化构造函数
alert(a.name);              //调用构造函数的实例对象
```

在 JavaScript 中，构造函数实际上就是类的一种抽象结构，可以把它作为类来看待和使用，但是 JavaScript 语言并没有定义类的概念。

利用 new 运算符来克隆 JavaScript 预定义的构造函数，可以快速生产出很多具体的、可以随时调用的实例函数对象。基本语法如下：


```
var f = new Function(p1, p2, ..., pn, body);
```

其中构造函数 `Function()` 的参数类型都是字符串, `p1~pn` 表示所创建函数的参数名称列表, `body` 表示所创建函数的函数结构体语句, 在 `body` 语句之间通过分号进行分隔。f 就是所创建函数的名称。例如:

```
var f = new Function("a", "b", "return a+b");
```

同样是定义函数结构, 上面代码也可以使用 `function` 语句来实现:

```
function f(a, b){
    return a + b;
}
```

可以不指定任何参数, 创建一个空函数结构体。例如:

```
var f = new Function();
```

也可以不指定函数名, 克隆一个匿名函数, 当然这样的函数也就没有实用价值, 因为无法进行读写:

```
new Function("a", "b", "return a+b");
```

注意, `p1~pn` 是参数名称列表, `p1` 不仅能代表一个参数, 它也可以是一个逗号隔开的参数列表。例如, 下面的定义方法是等价的:

```
var f = new Function("a", "b", "c", "return a+b+c");
var f = new Function("a, b, c", "return a+b+c");
var f = new Function("a,b", "c", "return a+b+c");
```

使用 `new Function()` 的形式来创建一个函数不是很常见, 因为一个函数体通常会有多条语句。如果将它们以一个字符串的形式作为参数传递, 代码的可读性会很差。JavaScript 引入 `Function` 类型并提供 `new Function()` 这样的语法, 是因为函数对象添加属性和方法就必须借助于 `Function` 这个类型。函数的本质是一个内部对象, 由 JavaScript 解释器决定其运行方式。通过上述代码创建的函数, 在程序中可以使用函数名进行调用。

构造函数一般没有返回值, 它们只是初始化由 `this` 关键字所指代的对象, 并且什么都不返回。但是, 构造函数可以返回一个对象, 返回的对象将成为 `new` 运算符的运算值, 此时 `this` 所引用的对象就会被覆盖。

例如:

```
function F(){
    this.x = 1;
    return { y : 2 };
}
var f = new F();
alert(f.x);
alert(f.y);
```

上面示例演示了如何使用返回的对象覆盖构造函数的实例对象, 但是如果返回值是原始值时, 就不会覆盖实例化对象。此时按照普通函数的方式调用构造函数, 就可以得到返回值。例如:

```
function F(){
    this.x = 1;
    return true;
}
var f = new F();
alert(f.x);
alert(F());
```

因此, 对于构造函数的返回值为对象的情况, 可以直接调用构造函数来引用该返回值对象, 而不需要使用 `new` 运算符来运算。例如, 针对上面示例, 返回值为对象 `{ y : 2 }`, 可以直接调用构造函数来引用该返回值对象:

```
var f = F();
alert(f.x);
alert(f.y);
```

使用 `Function()` 构造函数比使用 `function` 语句定义函数的优点如下:

- ☑ 使用 `Function()`构造函数可以动态创建和编译一个函数，它不会把开发人员限制在 `function` 语句预编译的函数结构体中。当然它也有一个缺点，就是每次调用函数时，`Function()`构造函数都要对其进行编译。如果在循环结构或者经常调用函数时，就会影响执行效率。
- ☑ 使用 `Function()`构造函数，而不是使用 `function` 语句定义函数，能够把函数当作表达式来使用，而不是当作一个结构固定的语句，因此使用起来就会更加灵活。

16.1.2 函数直接量

直接量实际上就是常量，函数直接量就是结构固定的函数体。例如：

```
function(a, b){           //函数直接量
    return a + b;
}
```

在上面代码中，函数直接量与使用 `function` 语句定义函数结构在语法上比较类似，它们的结构都是固定的。但是函数直接量没有指定函数名，而是直接利用关键字 `function` 来表示函数的结构，也称之为匿名函数。匿名函数完全可以看做是一个表达式，而不是作为语句。例如，在下面示例中把匿名函数作为一个直接量参与运算：

```
//把函数作为一个值直接赋值给变量 f
var f = function(a, b){
    return a + b;
};
```

当把函数结构当作一个值赋值给变量之后，此时该变量就具备函数的结构和功能，因此可以像使用函数那样来引用该变量：

```
alert(f(1,2)); //返回数值 3
```

实际上，可以把函数作为一个运算元，直接参与表达式运算。例如，对于上面示例可以使用如下代码来完成：

```
//把函数作为一个运算元，利用函数调用运算符 (()) 进行调用
alert(
    (function(a, b){
        return a + b;
    })(1,2)
); //返回数值 3
```

函数是可以相互嵌套的，因此可以定义复杂的嵌套结构体函数。例如：

```
function f(x, y){           //外层函数
    function e(a, b){       //内层函数
        return a * b;
    }
    return x + y;
}
```

嵌套的函数只能够在函数体内部自由使用，外部无权引用。内层函数除了可以参与函数体内表达式运算外，没有任何实际价值，所以 JavaScript 不提倡这种用法。例如：

```
function f(x, y){
    function e(a, b){
        return a * b;
    }
    return e(3, 6) + y;      //内层函数参与表达式运算有效
    alert(e(3, 6));          //无效的调用
}
alert(f(3, 6));              //调用外层函数
```

JavaScript 对于使用 `function` 语句定义的函数有一定的限制。例如，不允许在分支结构或循环结构中声明函数，仅允许在顶层全局代码或者顶层函数体内声明函数。不过把匿名函数作为数据参与函数内表达式运算则是 JavaScript 开发中常用的手法，其使用位置没有任何限制。

16.1.3 选择恰当的方法

虽然定义函数的结构体相同，函数的效果相近，但是选用不同的方法也比较讲究，详细比较如表 16.1 所示。

表 16.1 函数定义方法比较

属 性	使用 <code>function</code> 语句	使用 <code>Function()</code> 构造函数	使用函数直接量
兼容	完全	JavaScript 1.1 及以上	JavaScript 1.2 及以上
形式	句子	表达式	表达式
名称	有名	匿名	匿名
主体	标准语法	字符串	标准语法
性质	静态	动态	静态
解析	以命令的形式构造一个函数对象	解析函数体，能够动态创建一个新的函数对象	以表达式的形式构造一个函数对象
作用域	具有函数作用域	顶级函数，具有顶级作用域	具有函数作用域

1. 从函数作用域角度选择

使用 `Function()`构造函数创建的函数具有顶级作用域，JavaScript 解释器也总是把它作为顶级函数来编译，而 `function` 语句和函数直接量定义的函数都有自己的作用域（即局部作用域，或称函数作用域）。例如：

```
var n = 1;           //全局变量，作用域为当前文档
function f(){
    var n = 2;       //局部变量，作用域仅限于函数体内
    function e(){    //使用 function 语句定义的函数结构体
        return n;    //检测变量 n 到底返回什么值
    }
    return e;        //返回函数结构
}
alert(f())();        //返回 2，调用函数 f 的返回函数 e
```

在上面示例中，分别在函数体外和函数体内声明并初始化变量 `n`，然后在函数体内使用 `function` 语句定义一个函数 `e`，定义该函数返回变量 `n` 的值，最后在函数体外调用函数的返回函数。结果发现返回值为局部变量 `n` 的值（即为 2），也就是说 `function` 语句定义的函数拥有自己的作用域。

同理，如果使用函数直接量定义函数 `e`，当调用该返回函数时，返回的值是 2，而不是 1，也说明函数直接量定义的函数拥有自己的作用域。代码如下：

```
var n = 1;           //全局变量，作用域为当前文档
function f(){
    var n = 2;       //局部变量，作用域仅限于函数体内
    var e = function(){ //使用函数直接量定义的函数结构体
        return n;    //检测变量 n 到底返回什么值
    }
    return e;        //返回函数结构
}
alert(f())();        //返回 2，调用函数 f 的返回函数 e
```

但是如果使用 `Function()`构造函数定义函数 `e`，则调用该返回函数时，返回的值是 1，而不再是 2，因为

Function()构造函数定义的函数作用域需要动态确定，而不是在定义函数时确定的。代码如下：

```
var n = 1; //全局变量，作用域为当前文档
function f(){
    var n = 2; //局部变量，作用域仅限于函数体内
    var e = new Function("return n;");
    //使用 Function 构造函数定义的函数结构体
    return e; //返回函数结构
}
alert(f())(); //返回 1，调用函数 f 的返回函数 e
```

2. 从解析机制选择

JavaScript 解释器在解析代码时，并非一行一行地分析和执行程序，而是一段一段分析执行。在同一段代码中，使用 function 语句和函数直接量定义的函数结构总会被提取出来优先执行。只有当函数被解析和执行完毕之后，才会按顺序执行其他代码行。但是使用 Function()构造函数定义的函数并非提前运行，而是在运行时动态地被执行，这也说明了为什么 Function()构造函数定义的函数具有顶级作用域的根本原因。

因此，从时间的角度瞬间来审视，function 语句和函数直接量定义的函数具有静态的特性，而 Function()构造函数定义的函数具有动态的特性。这种解析机制的不同，必然带来不同的执行效率，这种差异性可以通过一个循环结构放大比较出来。例如，在下面这个示例中，分别把 function 语句定义的空函数和 Function()构造函数定义的空函数放在一个循环体内，让它们空转十万次，则会明显感到使用 function 语句定义的空函数运行效率高。

```
//测试 function 语句定义的空函数执行效率
var a = new Date(); //定义当前时间对象实例
var x = a.getTime(); //获取当前时间的毫秒数
for(var i=0;i<100000;i++){ //定义的循环结构体
    function(){ //使用 function 语句定义的空函数
        ;
    }
}
var b = new Date(); //定义当前时间对象实例
var y = b.getTime(); //获取当前时间的毫秒数
alert(y-x); //返回 62，不同环境和浏览器会存在差异

//测试 Function()构造函数定义的空函数执行效率
var a = new Date(); //定义当前时间对象实例
var x = a.getTime(); //获取当前时间的毫秒数
for(var i=0;i<100000;i++){ //定义的循环结构体
    new Function(); //使用 Function()构造函数定义的空函数
}
var b = new Date(); //定义当前时间对象实例
var y = b.getTime(); //获取当前时间的毫秒数
alert(y-x); //返回 2390，不同环境和浏览器会存在差异
```

原来在执行循环结构之前，JavaScript 解释器首先把 function 语句定义的函数提取出来进行编译，这样每次循环执行该函数时，就不再从头开始重新编译该函数对象了。而 Function()构造函数定义的函数每次循环时都需要动态编译一次，这样效率就非常低。

3. 从灵活性选择

从兼容角度考虑，使用 function 语句定义函数不用考虑 JavaScript 版本问题，所有版本都支持这种方法。而 Function()构造函数只能够在 JavaScript 1.1 及其以上版本中使用，函数直接量仅能够在 JavaScript 1.2 及其以上版本中有效。当然，目前大部分用户都已经使用了支持 JavaScript 1.5 版本的浏览器，版本问题已经不是大问题了。

Function()构造函数和函数直接量定义函数方法有点相似，它们都是使用表达式来创建的，而不是使用语句创建，这也给它们带来很大的灵活性。当函数仅使用一次时，非常适合使用表达式的方法来创建。

由于 Function()构造函数和函数直接量定义函数不需要额外的变量，它们直接在表达式中参与运算，从而节省了资源，避免了使用 function 语句定义函数占用内存的弊端，也就是说这些函数运行完毕即被释放而不再占用内存空间。

当然，对于 Function()构造函数来说，由于定义函数的主体必须以字符串的形式来表示，使用这种方法定义复杂的函数就显得有点笨拙，很容易出现语法错误。但是函数直接量的主体使用标准的 JavaScript 语法，看来使用函数直接量是一种比较快捷的方法。

通过 function 语句定义的函数被称为命名式函数、声明式函数或者函数常量，而通过匿名方式定义的函数称为引用式函数或者函数表达式，而把赋值给变量的匿名函数称为函数对象，该变量也可以称为函数引用。

16.2 使用函数

定义函数实际上就是定义变量，是在内存中标识一块空间，构建函数结构体，其中函数名就相当于函数的地址标识名。下面就来详细讲解函数的一般用法。

16.2.1 函数调用

调用函数就是执行函数，而引用函数就是传递函数的引用地址。

定义的函数在默认状态下是不会被执行的，一般使用函数调用运算符（()）来激活函数运行，在小括号运算符中可以包含 0 个或多个参数，参数之间通过逗号进行分隔（详细语法可以参阅前面章节说明）。例如：

```
function f(x,y){           //定义函数
    return x*y;           //返回值
}
alert(f(f(5,6),f(7,8)));   //返回 1680。重复调用函数
```

在上面示例中通过在函数中调用函数的方法实现多重调用，实际上就是把函数调用作为一个表达式的值直接进行传递，这样就节省了两个临时变量，可以把它转换为：

```
function f(x,y){           //定义函数
    return x*y;           //返回值
}
var a = f(5, 6);           //返回 30，调用函数
var b = f(7, 8);           //返回 56，调用函数
alert(f(a, b));           //返回 1680，调用函数
```

如果函数返回值为一个函数，则在调用时需要使用多个小括号运算符进行反复调用。例如：

```
function f(x, y){          //定义函数
    return function(){     //返回函数类型的数据
        return x * y;
    }
}
alert(f(7, 8)());          //返回值 56，反复调用函数
```

再看一个示例，在下面代码中定义函数的返回值为函数自身，从而设计一种递归返回函数自身的结构，调用该函数时，可以通过无数个小括号运算符进行反复调用，但是最终返回值都是函数结构体自身。

```
function f(){              //定义函数
    return f;              //返回函数自身
}
```

```
alert(f()()()()()()()()()()()); //返回函数结构体
```

另外, JavaScript 函数将遵循从内到外的原则就近调用, 但是不会从外到内调用下面的函数。这样就避免了嵌套结构中调用同名函数而引发的冲突问题。例如:

```
function f(){ //顶级函数 f
    return 1;
}
function o(){ //函数作用域
    return o() //调用内部函数 o
    function o(){ //函数内部作用域
        return f(); //嵌套函数内部函数 f
        function f(){ //嵌套函数内部函数 f
            return 3;
        }
    }
    function f(){ //嵌套函数 f
        return 2;
    }
}
alert( f() ); //返回数值 1
alert( o() ); //返回数值 3
```

在上面示例中, 在全局作用域内调用函数 `f`, 则将调用最顶级函数 `f`。同样, 在全局作用域内调用函数 `o`, 将调用最顶级函数 `o`。当调用顶级函数 `o` 时, 激活内部脚本并返回调用内部函数 `o`, 继续激活并调用最里层的函数 `f`。如果没有最里层的函数 `f`, 则将向上搜索函数 `f`, 并将调用嵌套函数 `f`, 返回数值 2。如果还没有检索到函数 `f`, 则将调用顶层函数 `f`, 返回数值 1。

16.2.2 生命周期

JavaScript 解释器在解析程序时, 会以段为单位逐段解析, 这个段表示的就是以 `<script>` 标签包含的脚本。在程序段内, 解释器会先预编译声明的变量和函数结构, 然后再逐行执行代码。但是对于匿名函数, 则会在代码执行进行编译, 并开始其生命周期。例如:

```
//第 1 个函数
function f(){ //函数 1, 使用 function 语句声明函数
    return 1;
}
alert(f()); //返回值为 4, 说明第 1 个函数被第 4 个函数覆盖
var f = new Function("return 2"); //函数 2, 使用构造函数定义函数
alert(f()); //返回值为 2, 说明第 4 个函数被第 2 个函数覆盖
var f = function(){ //函数 3, 使用函数直接量定义函数
    return 3;
}
alert(f()); //返回值为 3, 说明第 2 个函数被第 3 个函数覆盖
//第二波函数
function f(){ //函数 4, 使用 function 语句声明函数
    return 4;
}
alert(f()); //返回值为 3, 说明第 4 个函数被第 3 个函数覆盖
var f = new Function("return 5"); //函数 5, 使用构造函数定义函数
alert(f()); //返回值为 5, 说明第 3 个函数被第 5 个函数覆盖
var f = function(){ //函数 6, 使用函数直接量定义函数
    return 6;
}
```

```

}
alert(f());           //返回值为 6，说明第 5 个函数被第 6 个函数覆盖

```

通过上面的示例比较，可以看到：在代码段中，使用 `function` 语句声明的函数被预先编译，此时如果存在相同的函数名，则会按顺序被覆盖。而函数直接量和构造函数定义的函数都属于表达式函数，因此在代码按行执行时，才被激活其生命周期。

函数内部成员是在函数被处理时定义的，如参数变量、函数体内使用 `var` 语句声明的变量等。但是当函数调用完毕，即执行 `return` 语句之后，将释放所有资源。对于 `function` 语句声明的函数结构来说，依然保存着该函数在预编译时占据的内存空间。但是对于匿名函数来说，则将被完全释放。不过对于函数直接量来说，由于它是静态函数，会在系统中保存一份函数结构的备份，以备下次再次调用。

当然，在某种情况下，函数的运行域不一定完全被注销。由于函数本身就是一种类型的数据，当它被调用时，会生成一个临时的调用对象。一般情况下，函数调用结束后会自动注销这些临时变量，因此一次调用之后，函数调用时所初始化的局部变量和参数都将不复存在。不过，如果函数在注销之前，还被外部引用，这时函数结构体将会保持不变，因此就会存在函数调用完毕依然存在的情况。例如：

```

var a = [];
for(var i = 0; i<10;i++){
    (function(){
        a[i] = i*j;
    })(i);           //函数以闭包的形式存储循环变量，调用后不会被注销
}
alert(a);           //返回值字符串 0,1,4 ,9,16 ,25,36 ,49, 64,81

```

16.2.3 形参和实参

函数结构虽然是一种比较封闭的代码实体，但是它有两个接口。

- ☑ 进口：用来接收各种数据，这些被接收的数据就称为参数。
- ☑ 出口：出口能够把接收的数据或默认数据（没有参数）处理并返回，即称为函数的返回值。

形参就是形式上的参数，而实参就是实际传递的参数。例如：

```

function f(a,b){           //定义函数结构，传递形参 a 和 b
    return a+b;
}
var x=1,y=2;               //定义参数变量
alert(f(x,y));             //调用函数并传递实参

```

在上面示例中，函数结构中的变量 `a`、`b` 就是形参，而在调用函数时向函数传递的变量 `x`、`y` 就是实参。

一个函数需要什么类型的参数，需要多少参数，这都由函数本身来决定。JavaScript 函数可以包含 0 个或多个形参。函数定义时的形参可以通过 `length` 属性来获取。例如，针对上面的函数，使用如下方法可以获取它的形参个数：

```

alert(f.length);           //返回 2，获取函数的形参个数

```

一般情况下，函数的形参和实参数量应该相同，但是 JavaScript 并没有要求形参和实参必须相同。在特殊情况下，函数的形参和实参数量可以不相同。

- ☑ 如果函数实参数量少于形参数量，那么多出来的形参的值就是 `undefined`。例如：

```

(function(a,b){           //定义函数，包含两个形参
    alert(typeof a);       //返回 number
    alert(typeof b);       //返回 undefined
})(1);                    //调用函数，传递一个实参

```

- ☑ 如果函数实参数量多于形参数量，那么多出来的实参就不能够通过形参标识符来访问，函数会忽略掉多余的实参。例如，在下面这个示例中，实参 3 和 4 就被忽略掉了。

```

(function(a,b){           //定义函数，包含两个形参

```

```

    alert(a);           //返回 1
    alert(b);           //返回 2
  })(1,2,3,4);         //调用函数，传递 4 个实参

```

在 JavaScript 程序中，实参数量少于形参的情况经常存在，这种形式主要存在于允许参数为默认值的函数中，其实有很多内部函数或方法也允许用户不为它们传递参数值。而形参数量少于实参的情况比较少见，这种形式一般在那些参数个数不是很确定的函数中。

形参与函数体内使用 var 语句声明的变量一样都属于局部变量，只有在函数被执行时才会被定义，一旦函数返回，它们就随同函数结构一同被注销。因此，形参一般不占用内存，只是在函数调用时才产生内存问题，而实参在调用前必须是已经分配内存的。

至于形参与实参是否共用或者独享内存单元，只能看调用时参数的传递方式，是传递的值类型数据还是传递的引用地址类型数据。如果是普通变量，或者是数组元素的实参，那么形参在接收值之前，系统会给形参分配新的内存单元，形参的变化与实参无关，这种传递就是所谓的传值。如果传递的是数组名或者是引用地址，那么实际传递的值就是一个地址值。形参接收值后，与实参所指的是同一个地址，它们会共享内存单元，这种传递就是所谓的传址。所以用户是无法指定参数的存储方式，而只能根据系统来决定。

16.2.4 参数对象 Arguments

为了方便管理实际参数，JavaScript 定义了 Arguments 对象。实际上 Argument 对象是一个伪数组，与 jQuery 对象类似，可以通过数组下标的形式获取该集合中传递给函数的参数值。

例如，在下面这个函数中，没有指定形参，但是在函数体内通过 Arguments 对象可以获取传递给该函数的每个实参值：

```

function f(){           //定义没有形参的函数
    for(var i = 0; i < arguments.length; i ++ ){ //循环读取函数的 Arguments 对象
        alert(arguments[i]); //显示指定下标的实参的值
    }
}
f(3, 3, 6);             //逐个显示每个传递的实参

```

使用 Arguments 对象应注意以下几个问题：

- ☑ Arguments 对象仅能够在函数体内使用，它仅作为函数体的一个私有成员而存在，因此可以通过点号运算符来指定 Arguments 对象所属的函数（代码如下所示）。不过由于 Arguments 对象在函数体内是唯一的和可指向的，所以一般会省略前置路径，直接引用 Arguments 对象的调用标识符 arguments。

```

function f(){           //定义没有形参的函数
    for(var i = 0; i < f.arguments.length; i ++ ){ //循环读取函数的 Arguments 对象
        alert(arguments[i]); //显示指定下标的实参的值
    }
}
f(3, 3, 6);             //逐个显示每个传递的实参

```

- ☑ 通过数组的形式来引用 Arguments 对象包含的实参值，如 arguments[i]，其中 arguments 表示对 Arguments 对象的实际引用，变量 i 是 Arguments 对象集合的下标值，从 0 开始，直到 arguments.length。其中，length 是 Arguments 对象的一个属性，表示 Arguments 对象包含的实参的个数。
- ☑ 从本质上来分析，Arguments 对象不是真正的 Array 对象，它是一个数据集合。因为它缺乏 Array 对象的数据结构和数组功能。例如，无法使用 for/in 结构遍历 Arguments 对象的实参值。

```

function f(){
    for(i in arguments){
        alert(arguments[i]);
    }
}

```



```

    }
}
f(3, 3, 6); //没有任何提示信息

```

如果使用 `typeof arguments` 表达式尝试检测 `Arguments` 对象的类型，则会返回 `object`（对象），所以它不能像 `Array` 一样使用 `push()` 和 `pop()` 等方法。

`Arguments` 对象中每个元素实际上就是一个变量，这些变量被用来存储调用函数时传递的实参值。通过 `arguments[]` 数组和已经命名的形参可以引用这些变量的值。当然，使用 `Arguments` 对象可以随时改变实参的值。

例如，在下面这个示例中把循环变量的值传递给 `Arguments` 对象元素，以实现动态改变实参的值。

```

function f(){
    for(var i = 0; i < arguments.length; i++){
        //遍历 Arguments 对象元素
        arguments[i] = i;           //修改每个实参的值
        alert(arguments[i]);         //提示修改的实参值
    }
}
f(3, 3, 6);                        //返回提示 1、2、3，而不是 3、3、6

```

当然通过修改 `Arguments` 对象的 `length` 属性值，可以达到改变函数实参个数的目的。当 `length` 属性值增大时，增加的实参值为 `undefined`；如果 `length` 属性值减小，则会丢弃 `arguments` 数据集后面相对应个数的元素。例如：

```

function f(){
    arguments.length = 2;           //修改 Arguments 对象的 length 属性值
    for(var i = 0; i < arguments.length; i++){
        alert(arguments[i]);
    }
}
f(3, 3, 6);                        //返回提示 3、3

```

`Arguments` 对象在实际开发中具有重要的价值。开发人员使用它可以监测用户在调用函数时所传递的参数是否符合要求，增强函数的容错能力，同时还可以开发出很多功能强大的函数。

【示例 1】 增强函数的灵活性。如果要定义的函数参数个数允许不确定或者函数的参数个数很多，而又不想为每个参数都定义一个变量，此时定义函数时可以保留参数列表为空，在函数内部使用 `Arguments` 对象来访问调用函数时传递的所有参数。本示例利用 `Arguments` 对象来计算函数任意多个参数的平均值。

```

function avg(){                     //求平均数
    var num = 0, l = 0;             //声明并初始化临时变量
    for(var i = 0; i < arguments.length; i++){ //遍历所有实参
        if(typeof arguments[i] != "number") //如果参数不是数值
            continue;                 //则忽略该参数值
        num += arguments[i];          //计算参数的数值之和
        l++;                          //计算参与和运算的参数个数
    }
    num /= l;                       //求平均值
    return num;                     //返平均值
}
alert(avg(1, 2, 3, 4));             //返回 2.5
alert(avg(1, 2, "3", 4));           //返回 2.3333333333333335

```

【示例 2】 检测函数传递参数的合法性。表单验证是页面设计中常做的任务，下面示例验证所输入的值是否符合邮箱地址格式。

```

function isEmail(){
    if(arguments.length>1) throw new Error("只能传递一个参数"); //检测参数个数
    //定义正则表达式

```

```

var regexp = /^w+((-w+))(\.w+))*@[A-Za-z0-9]+((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$/;
if (arguments[0].search(regexp)!= -1)      //匹配实参的值
    return true;                          //如果匹配则返回 true
else
    return false;                        //如果不匹配则返回 false
}
var email = "zhuyinhong@css8.cn";          //声明并初始化邮箱地址字符串
alert(isEmail(email));                    //返回 true

```

16.2.5 回调函数 callee

Arguments 对象包含一个 callee 属性，它能够返回当前 Arguments 对象所属的函数引用，这相当于在函数体内调用函数自身（引用函数地址）。callee 属性比较有用，特别是在匿名函数中，通过它在函数内部来引用函数自己。

例如，在下面这个示例中，通过 arguments.callee 获取对当前匿名函数的引用，然后通过函数的 length 属性确定它的形参个数，最后通过实参和形参数目比较来确定传递的参数是否合法。

```

function f(x, y, z){
    var a = arguments.length;          //获取函数实参的个数
    var b = arguments.callee.length;   //获取函数形参的个数
    if (a != b){                        //如果形参和实参个数不相等，则提示错误信息
        throw new Error("传递的参数不匹配");
    }
    else{                               //如果形参和实参数目相同，则返回它们的和
        return x + y + z;
    }
}
alert(f(3, 4, 5));                    //返回值为 12

```

注意，Function 对象的 length 属性返回的是函数的形参个数，而 Arguments 对象的 length 属性返回的是函数的实参个数。

如果函数不是匿名函数，则 arguments.callee 等价于函数名。例如，上面示例可以修改为如下形式：

```

function f(x, y, z){
    var a = arguments.length;          //获取函数实参的个数
    var b = f.length;                  //在函数体内通过函数名获取函数形参的个数
    if (a != b){                        //如果形参和实参个数不相等，则提示错误信息
        throw new Error("传递的参数不匹配");
    }
    else{                               //如果形参和实参数目相同，则返回它们的和
        return x + y + z;
    }
}
alert(f(3, 4, 5));                    //返回值为 12

```

16.2.6 返回值

函数使用 return 语句返回一个值，它相当于函数的出口，作为函数与外界进行交流的一个主要通道。在函数体内，可以定义 return 语句，也可以省略 return 语句。一旦执行 return 语句，它将停止函数的运行，并且把表达式的值返回给函数引用。如果函数不包含 return 语句，则函数体内每条语句被执行完毕之后，返回 undefined 值。

由于 JavaScript 是一种弱类型的语言，所以函数对于接收和输出数据都没有数据类型的限制，JavaScript 也不会自动检测输入和输出数据的类型。如果需要检测输入数据的类型，可以通过 `typeof` 运算符来进行检测，具体示例可以参阅函数参数节内容。因此，可以定义函数的返回值为函数等复杂结构的数据类型。

```
function f(){
    return function(x, y){           //返回值为函数
        return x + y;
    }
}
```

同时对于接收的数据个数也没有限制，但是返回的值只能够有一个。如果要输出多个数值，则可以通过数组的形式进行传递。例如：

```
function f(){
    var a = [];
    a[0] = true;
    a[1] = function(x, y){
        return x + y;
    }
    a[2] = 123;
    return a;                       //返回多个值
}
```

一个函数可以包含多个 `return` 语句，但是一般仅能够执行第 1 个 `return` 语句，因此常在函数体内通过分支结构来决定函数的返回选择项。例如：

```
function f(x, y){                   //根据条件返回值
    if(x > y) return x - y;
    if(x < y) return y - x;
    if(x * y <= 0) return x + y;
}
```

16.3 函数对象

在 JavaScript 中，函数是一类数据。但是，如果使用 `Function` 构造函数来创建函数，函数就继承了 `Function` 对象的所有信息，此时函数就拥有 `Function` 对象特性，可以调用它的属性和方法，同时可以自定义自己的属性和方法。

在函数结构体内，函数默认定义了 `Arguments` 对象，该对象包含一个 `length` 属性，利用它可以确定函数实参的个数。但是函数（`Function`）对象本身也定义了一个 `length` 属性，它可以返回函数定义时所指定的形参个数，不过这个属性是一个只读属性。与 `Arguments` 对象的 `length` 属性不同，`Function` 对象的 `length` 属性在函数结构体内外都可以使用。例如：

```
function f(x,y,z){                 //定义包含 3 个形参的空函数 f
    alert(f.length);              //返回 3，而 Arguments 对象的 length 属性仅能够在函数体内部使用
    function check( a ){          //定义检测函数实参与形参是否一致的功能函数
        if( a.length != a.callee.length ) //如果实参与形参的 length 属性值不同，则抛出错误
            throw new Error( "参数不一致" );
    }
    function f( a, b, c, d ){      //定义一个普通应用函数
        check( arguments );       //调用函数 check
        return ( a + b + c + d ) / 3; //返回函数值
    }
    alert( f( 3, 4 ) );           //抛出异常。调用函数 f，传递两个参数
```


函数作为对象不仅预定义了很多属性和方法，同时也允许用户自定义函数属性和方法，从而可以自由增强函数的功能。注意，函数的属性和方法与对象的属性和方法还是存在很多不同的。虽然从大的结构来分析，它们都属于同一个祖宗，但是在定义方法上还存在差异。定义函数的属性和方法通过点号运算符来实现：

```
function.property
function.method
```

函数属性可以在函数结构体内定义，也可以在函数体外定义。例如：

```
function f(){
    f.x=1;           //在函数体内定义属性
}
f.y=2;              //在函数体外定义属性
```

函数外定义的属性可以任意地访问，可以在函数体内（即函数作用域内部）或者外部（全局作用域），而函数内部定义的属性在初始化状态只能够在函数体内部被调用。例如：

```
function f(){
    f.x=1;           //在函数体内定义属性
    alert(f.x)       //返回 1，在函数体内调用
    alert(f.y)       //返回 2，在函数体内调用
}
f.y=2;              //在函数体外定义属性
alert(f.x)          //返回 undefined，在函数体外调用无效
alert(f.y)          //返回 2，在函数体外调用有效。不过如果函数被调用之后，其定义的属性可以随意调用。例如：
function f(){
    f.x=1;           //在函数体内定义属性
    alert(f.x)       //返回 1，在函数体内调用
    alert(f.y)       //返回 2，在函数体内调用
}
f.y=2;              //在函数体外定义属性
alert(f.x)          //返回 undefined，在函数体外调用无效
alert(f.y)          //返回 2，在函数体外调用有效
```

如果说属性是函数的私有变量，那么方法则是函数的私有函数。注意，它与嵌套函数在性质上是不同的。因此，函数属性和方法的定义方法和用法也基本相同，要注意方法调用的时机和场合。例如：

```
function f(){
    f.x=function(x){ //在函数体内定义函数的方法 x()
        return x
    };
    alert(f.x(4));   //返回 4，在函数体内调用函数的方法 x()
    alert(f.y(4));   //返回 16，在函数体内调用函数的方法 y()
}
f.y=function(y){    //在函数体外定义函数的方法 y()
    return y*y;
};
f();                //调用函数 f()
alert(f.x(4));      //返回 4，在函数体外调用函数的方法 x()
alert(f.y(4));      //返回 16，在函数体外调用函数的方法 y()
```

经常在开发中，希望在函数内能够定义一个这样的变量，它的值能够随着函数的反复调用而传递不同的值，从而实现在循环结构中函数可以执行不同的任务。例如：

```
function f(){        //定义函数
    var x = 0;        //定义局部变量并赋值为 0
    return x++;        //希望每次调用函数时能够返回不同的值
```



```
}  
for( var i = 1; i < 10; i ++ ){    //循环结构中反复调用函数 f()  
    alert( f() );                //总是返回 0  
}
```

上面示例并非每次都返回不同的值，原因是局部变量的值在每次调用函数时，都被重新初始化。当然可以通过一个全局变量来实现这个想法。代码如下：

```
var x = 0;                        //定义全局变量并赋值为 0  
function f(){                    //定义函数  
    return x++;                  //希望每次调用函数时能够返回不同的值  
}  
for( var i = 1; i < 10; i ++ ){  //循环结构中反复调用函数 f()  
    alert( f() );                //每次返回的值都不同  
}
```

但是这种设计方法，很明显缺乏封闭性，函数调用全局变量来实现自己的任务，在大型应用中这是非常危险的举措。所以，最安全的方法是函数定义属性，然后利用属性来实现函数每次都返回不同的值：

```
function f(){                    //定义函数  
    return f.x++;                //返回函数 f() 的属性 x 的递增值  
}  
f.x = 0;                         //定义函数的属性 x，并初始化为 0  
for( var i = 1; i < 10; i ++ ){  
    alert( f() );  
}
```

上面示例就能够很好地实现上述所设计的意图，同时也确保了函数结构的封闭性。

16.4 动态指针

`this` 是一个非常灵活的动态指针，它始终指向当前函数的调用对象。`this` 使用范围仅局限于函数内或者调用范围内。具体用法如下：

```
this[.属性]
```

如果不包含属性参数，则传递的是当前对象。

16.4.1 认识 this

`this` 就是一个指针，一个具有动态特性的指针，在 JavaScript 中类似指针特性的标识符还有以下 3 个。

- ☒ `callee`：函数的参数集合包含的一个静态指针，它始终指向参数集合所属的函数。
- ☒ `prototype`：函数包含的一个半静态指针，在默认状态下它始终指向函数附带的原型对象。不过可以改变这个指针，使它可以指向其他对象。
- ☒ `constructor`：对象包含的一个指针，它始终指向创建该对象的构造函数。

不过，`this` 指针是非常灵敏的，它所指代的对象是由 `this` 所在的执行作用域决定的，而不是根据 `this` 所在的定义作用域决定（如图 16.1 所示）。因此，可以把 `this` 看做是 JavaScript 执行作用域的一个属性，这个属性始终引用当前上下文环境对象。

JavaScript 函数可以在不同环境中被引用，既可以作为一个对象的方法，也可以作为另一个对象的方法，而且这种引用是在执行时才确定。因此，JavaScript 方法的灵活性注定了 `this` 指针只能在运行环境中动态地确定。

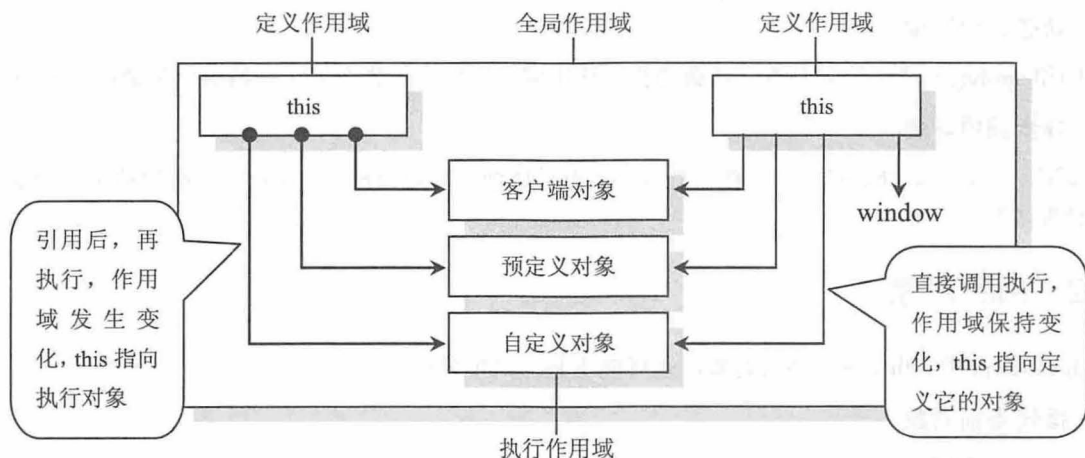


图 16.1 this 指针的变化示意图

1. 闭包环境

在 JavaScript 中，闭包扮演着非常重要的角色，它能够改变 this 指针的指代对象。例如，下面这个示例中，一个对象中（如 o1）引用或调用另一个对象（如 o）的方法时，该方法中的 this 指针是变化的。当引用对象 o 的方法 f() 时，它的 this 就会根据执行对象而定。但是如果调用对象 o 的方法 f() 时，它的 this 还是指向原定义的对象 o。

```
var name = "this = window";
var o = {
  name: "this = o",
  f: function(){
    return this;
  }
};
var o1 = {
  name: "this = o1",
  f: o.f //引用对象 o 中的方法 f()
}
var a = o1.f();
alert(a.name); //this 实际指向对象为 o1
下面把对象 o 的方法 f()封装在闭包中，然后再进行引用：
var o1 = {
  name: "this = o1",
  f: function(){ //this 使用闭包封装方法的引用
    return o.f;
  }
}
var a = o1.f();
alert(a.name); //this 实际指向对象为 Window
```

这时会发现，方法 f() 中的 this 既不指向对象 o，也不指向对象 o1，而是指向对象 Window。因为在执行对象 o1 的方法 f() 时，仅是指定了闭包运行的上下文环境，即执行闭包的对象为 o1，但是闭包内包裹的方法 f() 并没有被执行，而再次执行闭包内的方法时，此时执行环境已经发生了变化，执行对象从 o1 变为全局对象 Window，所以 this 就指向了 Window。

2. 动态调用环境

call()和 apply()方法能够强制改变函数的执行作用域，所以它们也会破坏函数引用和调用的一般规律。

3. 异步调用环境

异步调用也会破坏 this 指针变化的一般规律，而且这种情况更为神秘，这是因为函数被传递给定时器或者事件处理函数时才被调用。

16.4.2 this 对象

在 JavaScript 中，this 指代当前对象，也可能不是。简单分析如下。

1. 指代当前对象

☑ 指代当前操作对象

下面这个按钮定义了一个单击事件属性，其中就包含了 this 关键字。代码如下：

```
<input type="button" value="主人是谁？" onclick = "this.value = '我就是主人'" />
```

其中，onclick 事件属性中包含的 this 就代表当前对象 input。

☑ 指代构造函数实例

定义一个构造函数，在其中使用 this 关键字作为临时代表，然后使用 new 运算符实例化构造函数。代码如下：

```
function F(){  
    this.name = "我就是主人";  
}  
var f = new F();  
alert(f.name);
```

显然，这里的 this 就代表当前实例对象 f。

☑ 指代当前对象直接量

下面是一个对象直接量，它包含了两个属性，其中方法 me()返回关键字 this。代码如下：

```
var o = {  
    name : "我是对象 o",  
    me : function(){  
        return this;  
    }  
}  
var who = o.me(); //调用对象 o 的方法 me()  
alert(who.name); //读取 this 所代表对象属性 name，返回字符串“我是对象 o”
```

当调用对象直接量 o 的方法 me()后，变量 who 的值就是 this 的值，它代表当前对象直接量 o，然后读取对象 o 的属性 name，则返回字符串“我是对象 o”。

☑ 指代全局对象

在函数 f()中调用 this 关键字，并为 this 定义并初始化一个属性 name，当调用函数之后，则可以直接读取属性 name。代码如下：

```
function f(){  
    this.name = "我是谁？";  
}  
f(); //调用方法 f()  
alert(name); //直接读取属性 name，返回值为“我是谁？”
```

方法 f()是全局对象 window 方法，this 实际上就是代表 window，而全局对象可以省略，所以就看到上

面特殊的写法。实际上可以完整地写成如下这样：

```

window.f = function(){
    this.name = "我是谁？"
}
window.f();
alert(window.name);    //读取客户端全局对象 window 的属性 name，返回值为“我是谁？”
如果进一步修改，会看到实际上它与对象直接量在用法上有几分相似之处：
var window = {
    f : function(){
        this.name = "我是谁？";
    }
}
window.f();
alert(window.name);    //读取对象直接量 window 的属性 name，返回值为“我是谁？”

```

2. 指代当前作用域

this 关键字并不总是代表当前对象，下面这个示例能够很好地说明这个问题：

```

var f = function(){
    alert(this);
}
f();                //直接调用函数 f()
new f();            //new 运算符调用函数 f()，即实例化对象

```

通过简单的比较会发现，直接调用函数 f()，返回的字符串为[object]。而使用 new 运算符调用函数 f() 之后，返回的字符串为[object Object]。使用 new 运算符实际上是把函数进行实例化，此时 this 指向当前实例对象。但是，直接调用函数 f()，则 this 表示 Window 对象。同一个语境中的 this 关键字，由于使用方式的不同，所返回的引用也是不同的。

实际上，这与 this 所在作用域有着密切关系。函数 f() 属于全局作用域，也就是说，它应该属于 Window 对象。如果直接调用函数 f()，当然函数中的 this 关键字就代表 Window 对象了。而使用 new 运算符实例化函数 f() 后，也就创建了一个新对象，当前作用域就不再是全局作用域了，而是对象作用域，所以 this 就代表这个新创建的实例对象。再看一个示例：

```

<input type="button" value="主人是谁？" onclick =
    "this.value = '我就是主人'" />
<input type="button" value="主人是谁？" onclick = "f()" />
<script language="javascript" type="text/javascript">
function f(){
    this.value = "我在哪儿？";
}
</script>

```

该按钮的事件属性中包含的 this 就是指向当前对象。但是，第 2 个按钮的鼠标单击事件属性中是以调用的方式调用全局作用域中的函数 f()，所以其中的 this 就代表 Window 对象，而不是当前按钮（第 2 个按钮）。

如果改变函数的用法。首先，在脚本中获取第 2 个按钮对象的引用，然后把函数 f() 作为一个值传递给对象的 onclick 事件属性，代码如下：

```

<input type="button" value="主人是谁？" />
<script language="javascript" type="text/javascript">
var btn2 = document.getElementsByTagName("input")[1];
btn2.onclick = f;
function f(){
    this.value = "我回来了";
}

```



```

}
</script>

```

在上面的示例中，`this` 关键字就表示按钮对象本身，所以当单击之后，也就能够改变按钮的值。这也说明当函数赋值给按钮对象之后，虽然函数的定义作用域没有发生变化，但是它的执行作用域已经从全局作用域变为对象作用域，所以 `this` 关键字所代表的对象也会随之发生变化。

如果以同样的方式，把该函数作为事件处理函数赋值给不同的按钮对象，则 `this` 会分别代表不同的按钮对象。这也说明如果改变函数的执行作用域，则函数所包含的 `this` 关键字也会指向不同的对象。再如：

```

function f(){
    return this;
}
var o={
    name:"对象 o",
    me:f,
    o1:{
        name:"对象 o1",
        me:f,
        o2:{
            name:"对象 o2",
            me:f
        }
    }
}
var who = o.o1.o2.me();
alert(who.name);           //返回字符串“对象 o2”，说明当前 this 代表对象 o2
var who = o.o1.me();
alert(who.name);           //返回字符串“对象 o1”，说明当前 this 代表对象 o1
var who = o.me();
alert(who.name);           //返回字符串“对象 o”，说明当前 this 代表对象 o

```

首先，定义函数 `f()`，设置其返回值为 `this` 关键字，然后把该函数作为值传递给不同作用域中的属性 `me`，最后分别调用它们，会发现 `this` 所代表的对象是不同的。

但是，如果不是以值的方式传递函数 `f()`，而是直接调用，则它们所包含的 `this` 都代表 `Window` 对象。也就是说，`this` 的执行作用域并没有发生变化，仍然根据定义它的作用域来确定，即全局作用域。代码如下：

```

var name = "我是 Window 对象";
function f(){
    return this;
}
var o={
    name:"对象 o",
    me:f(),
    o1:{
        name:"对象 o1",
        me:f(),
        o2:{
            name:"对象 o2",
            me:f()
        }
    }
}
var who = o.o1.o2.me;

```

```

alert(who.name);           //返回字符串“我是 Window 对象”，说明 this 代表对象 Window
var who = o.o1.me;
alert(who.name);           //返回字符串“我是 Window 对象”，说明 this 代表对象 Window
var who = o.me;
alert(who.name);           //返回字符串“我是 Window 对象”，说明 this 代表对象 Window

```

16.4.3 this 应用

this 总是指向当前作用域对象，如果当前定义对象的作用域没有发生变化，则它会指向当前对象。但是，this 关键字的用法比较灵活，同时 this 关键字可以存在于任何位置，它不仅局限于对象的方法内，还可以被应用在全局域内、函数内以及其他特殊上下文环境中。

1. 函数的引用和调用

函数的引用和调用表示不同的概念，虽然它们都无法改变函数的定义作用域。但是引用函数却能够改变函数的执行作用域，而调用函数是不会改变函数的执行作用域的。继续以上面示例为例进行说明：

```

var o = {
    name : "对象 o",
    f : function(){
        return this;
    }
}
o.o1 = {
    name : "对象 o1",
    me : o.f           //引用对象 o 的方法 f()
}

```

函数中的 this 所代表的是当前执行域对象 o1：

```

var who = o.o1.me();
alert(who.name);           //返回字符串“对象 o1”，说明当前 this 代表对象 o1

```

如果把对象 o1 的 me 属性值改为函数调用：

```

o.o1 = {
    name : "对象 o1",
    me : o.f()           //调用对象 o 的方法 f()
}

```

则函数中的 this 所代表的是定义函数时所在的作用域对象 o：

```

var who = o.o1.me;
alert(who.name);           //返回字符串“对象 o”，说明当前 this 代表对象 o

```

2. call()和 apply()方法

call()和 apply()方法可以直接改变被执行函数的作用域，使其作用域指向所传递的参数对象。因此，函数中包含的 this 关键字也指向参数对象。例如：

```

function f(){
    //如果当前执行域对象的构造函数等于当前函数，则表示 this 为实例对象
    if(this.constructor == arguments.callee) alert("this = 实例对象");
    //如果当前执行域对象等于 window，则表示 this 为 Window 对象
    else if (this == window) alert("this = window 对象");
    //如果当前执行域对象为其他对象，则表示 this 为其他对象
    else alert("this == 其他对象 \nthis.constructor = " +
this.constructor );
}

```

```
f();           //this 指向 Window 对象
new f();       //this 指向实例对象
f.call(1);     //this 指向数值实例对象
```

在这个示例中，直接调用函数 `f()` 时，函数的执行作用域为全局域，所以 `this` 代表 `Window`。当使用 `new` 运算符调用函数时，将创建一个新的实例对象，函数的执行作用域为实例对象所在的上下文，所以 `this` 就指向这个新创建的实例对象。

而使用 `call()` 方法执行函数 `f()` 时，`call` 会把函数 `f()` 的作用域强制修改为参数对象所在的上下文。由于 `call()` 方法的参数值为数字 1，则 JavaScript 解释器会把数字 1 强制封装为数值对象，此时 `this` 就会指向这个数值对象。

在下面这个示例中，`call()` 方法把函数 `f()` 强制转换为对象 `o` 的一个方法并执行，这样函数 `f()` 中的 `this` 就指代对象 `o`，所以 `this.x` 的值就等于 1，而 `this.y` 的值就等于 2，结果就返回 3。

```
function f(){           //函数 f()
    alert(this.x + this.y);
}
var o = {               //对象直接量
    x: 1,
    y: 2
}
f.call(o);              //执行函数 f(), 返回值为 3
```

3. 原型继承

JavaScript 通过原型模式实现类的延续和继承，那么在父类的成员中包含了 `this` 关键字时，当子类继承了父类的这些成员时，`this` 的指向就变得很迷惑人。一般情况下，子类继承父类的方法后，`this` 会指向子类的实例对象，但是也可能指向子类的原型对象，而不是子类的实例对象。例如：

```
function Base(){        //基类
    this.m = function(){ //基类的方法 m()
        return "Base";
    };
    this.a = this.m();    //基类的属性 a，调用当前作用域中的 m()方法
    this.b = this.m();    //基类的方法 b()，引用当前作用域中的 m()方法
    this.c = function(){ //基类的方法 c()，以闭包结构调用当前作用域中的 m()方法
        return this.m();
    }
}
function F(){           //子类
    this.m = function(){ //子类的方法 m()
        return "F"
    }
}
F.prototype = new Base(); //继承基类
var f = new F();         //实例化子类
alert(f.a);              //返回字符串 Base，说明 this.m()中 this 指向 F 的原型对象
alert(f.b());            //返回字符串 Base，说明 this.m()中 this 指向 F 的原型对象
alert(f.c());            //返回字符串 F，说明 this.m()中 this 指向 F 的实例对象
```

在上面的示例中，基类 `Base` 包含 4 个成员，其中成员 `b` 和 `c` 以不同方式引用当前作用域内的方法 `m()`，而成员 `a` 存储着当前作用域内方法 `m()` 的调用值。当这些成员继承给子类 `F` 后，其中 `m`、`b` 和 `c` 成为原型对象的方法，而 `a` 成为原型对象的属性。但是，`c` 的值为一个闭包体，当在子类的实例中调用时，实际上它的返回值已经成为实例对象的成员。也就是说，闭包体在哪儿被调用，则其中包含的 `this` 就会指向哪儿。所以，

会看到 `f.c()` 中的 `this` 指向实例对象，而不是 `F` 类的原型对象。

为了避免因继承关系而影响父类中 `this` 所代表的对象，除了通过上面介绍的方法，把方法的引用传递给父类的成员外，还可以为父类定义私有函数，然后再把它的引用传递给其他父类成员，这样就避免了因为函数闭包的原因而改变 `this` 的值。代码如下：

```
function Base(){
    var _m = function(){}           //定义基类的私有函数_m()
    return "Base";
};
this.a = _m;
this.b = _m();
}
```

这样基类的私有函数 `_m()` 就具有完全隐私性，外界其他任何对象都无法直接访问它。所以，在一般情况下，定义方法时，对于相互依赖的方法，可以把它定义私有函数，并以引用方法的方式对外公开，这样就避免了外界对于依赖方法的影响。

4. 异步调用之回调函数

异步调用就是通过事件机制或者计时器来延迟函数的调用时间和时机。不过调用函数的执行作用域不再是原来的定义作用域，所以函数中的 `this` 总是指向引发该事件的对象。例如：

```
<input type="button" value="Button" />
<script language="javascript" type="text/javascript">
var button = document.getElementsByTagName("input")[0];
var o = {};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
    if(this == button) alert("this = button");
}
button.onclick = o.f;
</script>
```

方法 `f()` 所包含的 `this` 不再指向对象 `o`，而是指向按钮 `button`，因为它被传递给按钮的事件处理函数之后，才被调用执行的。函数的执行作用域发生了变化，所以不再指向定义方法时所指定的对象。

如果使用 DOM 2 级标准为按钮注册事件处理函数：

```
if(window.attachEvent){           //兼容 IE
    button.attachEvent("onclick", o.f);
}
else{                             //兼容符合 DOM 标准的浏览器
    button.addEventListener("click", o.f, true);
}
```

则会看到，在 IE 浏览器中，`this` 指向 `Window` 和 `button`，而在符合 DOM 标准的浏览器中仅指向 `button`。因为，在 IE 浏览器中，`attachEvent()` 是 `Window` 对象的方法，调用该方法时，执行作用域为全局作用域，`this` 会指向 `Window`。同时由于该方法被注册到按钮对象上，所以它的真正执行作用域应该为 `button` 对象所在的上下文。这一点可以通过在符合 DOM 标准的浏览器中看到。

为了解决这个问题，可以借助 `call()` 或 `apply()` 方法强制在对象 `o` 身上执行方法 `f()`。也就是说，强制改变 `f()` 方法的执行作用域，避免因为环境的不同而影响函数作用域的变化。代码如下：

```
if(window.attachEvent){
    button.attachEvent("onclick", function(){           //以闭包的形式封装 call()方法强制执行 f()
        o.f.call(o);
    });
}
```



```

else{
    button.addEventListener("click", function(){
        o.f.call(o);
    }, true);
}

```

这样当再次预览时,方法 f()中包含的 this 关键字始终指向对象 o。也就是说,它的执行作用域始终与它的定义作用域保持一致。

5. 异步调用之定时器

异步调用的另一种形式就是使用定时器来调用函数,定时器就是指调用 Window 对象的 setTimeout()或 setInterval()方法来延期调用函数。例如,针对 16.4.2 节的示例,可以这样来设计延期调用方法 o.f():

```

var o = {};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
    if(this == button) alert("this = button");
}
setTimeout(o.f, 100);

```

此时,经测试程序,会发现在 IE 中 this 指向 Window 和 button 对象,具体原因与上面讲解的 attachEvent()方法相同。但是,在符合 DOM 标准的浏览器中, this 指向 Window 对象,而不是 button 对象,因为方法 setTimeout()是在全局作用域中被执行的,所以 this 自然指向 Window 对象。要解决这个问题,可以使用 call()或 apply()方法来实现:

```

setTimeout(function(){
    o.f.call(o);
}, 100);

```

16.4.4 this 陷阱

在 JavaScript 中, this 的用法是非常灵活的,只要理解 this 始终指向当前执行作用域所在的对象,那么所有问题也就迎刃而解了。当然,读者也必须掌握一些常规的应对策略,避免陷入 this 误区。

1. 基本原则

好的使用习惯对于预防 this 误解具有明显的效果。事实上, this 的复杂性很大程度上取决于用户的使用方式。由于 this 指代灵活,如果把它放在复杂的应用环境中,它也会变得很不确定。读者不妨记住一条:确保在同一域中操作包含 this 的方法或函数。应避免把包含 this 的全局函数或方法动态用在局部域的对象中,也应避免在不同作用域的对象之间相互引用包含 this 的方法或属性。

2. 把 this 作为参数值

如果把 this 作为参数值来调用函数,就可以避免 this 多变的问题,因为 this 始终与当前对象保持一致。例如,下面的做法是错误的,因为 this 在这里始终指向 Window 对象,而不是当前按钮对象。

```

<input type="button" value="按钮 1" onclick="f()" />
<input type="button" value="按钮 2" onclick="f()" />
<input type="button" value="按钮 3" onclick="f()" />
<script language="javascript" type="text/javascript">
function f(){
    alert(this.value);
}
</script>

```

但是，如果换一种思维，把 `this` 作为参数值进行传递，那么它就会代表当前对象：

```
<input type="button" value="按钮 1" onclick="f(this)" />
<input type="button" value="按钮 2" onclick="f(this)" />
<input type="button" value="按钮 3" onclick="f(this)" />
<script language="javascript" type="text/javascript">
function f(o){
    alert(o.value);
}
</script>
```

3. 设计静态的 `this` 指针

如果要确保构造函数的方法在初始化之后方法所包含的 `this` 指针不再发生变化，一个很简单的方法就是：在构造函数中把 `this` 指针存储在私有变量中，然后在方法中使用私有变量来引用 `this` 指针，这样所引用的对象始终都是初始化的实例对象，而不会在类型继承中发生变化。例如：

```
function Base(){                //基类
    var _this = this;           //存储初始化时对象的引用指针
    this.m = function(){
        return _this;          //返回初始化时对象的引用指针
    };
    this.name = "Base";
}
function F(){                    //子类
    this.name = "F";
}
F.prototype = new Base();        //继承基类
var f = new F();                 //实例化子类
var n = f.m();
alert(n.name);                   //this 始终指向原型对象，而不再是子类的实例对象
```

对于对象直接量来说，如果希望使用 `this` 代表当前对象直接量，则可以直接调用对象直接量的名称，而不用 `this` 关键字。例如：

```
var o = {
    name : "this = o",
    b : function(){
        return o;               //返回对象直接量名称，而不是 this
    }
}
var o1 = {
    name : "this = o1",
    b : o.b
}
var a = o1.b();
alert(a.name);
```

4. 设计静态的 `this` 扩展方法

当然，作为一个动态指针，`this` 也是可以被转换为静态指针的。实现的方法主要是利用 `Function` 对象的 `call()` 或 `apply()` 方法。在前面的示例中也已经提及它们的用法，这两个方法都可以强制指定 `this` 的指代对象。例如，为 `Function` 对象扩展一个原型方法 `pointTo()`，具体代码如下：

```
//把 this 转换为静态指针
//参数：o 表示预设置 this 所指代的对象
//返回值：返回一个闭包函数
Function.prototype.pointTo = function(o){
```

```

var _this = this;                                //存储当前函数对象
return function(){                               //返回一个闭包函数
    return _this.apply(o, arguments);
    //返回执行当前函数，并把当前函数的作用域强制设置为指定对象
}

```

这个方法将调用当前函数，并在指定的参数对象上执行，从而把 `this` 绑定到该对象上，然后应用这个函数扩展方法，以实现强制指定对象 `o` 的方法 `b()` 中的 `this` 始终指向定义对象 `o`。代码如下：

```

var o = {
    name : "this = o"
}
o.b = (function(){
    return this;
}).pointTo(o);                                //把 this 绑定到对象 o 身上
var o1 = {
    name : "this = o1",
    b : o.b
}
var a = o1.b();
alert(a.name);                                //返回字符串 this=o，说明 this 的值没有发生变化，始终指向对象，而不是对象 o1

```

还可以扩展 `new` 运算符的替代方法，从而间接使用自定义函数实例化类。代码如下：

```

//把构造函数转换为实例对象
//参数：f 表示构造函数
//返回值：返回构造函数 f() 的实例对象
function instanceFrom(f){
    var a = [].slice.call(arguments, 1);        //获取构造函数的参数
    f.prototype.constructor = f;                //手工设置构造函数的原型构造器
    f.apply(f.prototype, a); //在原型对象上强制指定构造函数，则原型对象就成为了构造函数的实例，同时由于它的构造器已经被设置为了构造函数，则此时原型对象就相当于一个构造函数的实例对象
    return f.prototype;                          //返回该原型对象
}

```

例如，下面的示例演示了如何使用这个自定义的实例化类方法来把一个简单的构造函数转换为具体的实例对象：

```

function F(){
    this.name = "F";
}
var f = instanceFrom(F);
alert(f.name);

```

通过这个示例也进一步说明，`call()` 和 `apply()` 方法具有强大的功能，它不仅能够执行普通函数，也能够实例化构造函数，担当 `new` 运算符的运算功能。

通过上面示例，也警示我们：`this` 非常灵敏，同时它也是一把双刃剑，用得好能够让 JavaScript 程序生辉，用不好会破坏整个程序的逻辑，所以在使用时应该异常慎重。最后再看一个嵌套函数结构体内 `this` 的变化：

```

function f(){
    this.a = " a ";                                //属于 Window 对象所有
    alert( this.a + this.b + this.c + this.d );    //返回 a undefined undefined undefined
    e();                                            //f 函数域内调用，属于 Window 对象所有
    function e(){
        this.b = " b ";
        alert( this.a + this.b + this.c + this.d ); //返回 a b undefined undefined
        g();                                        //e 函数域内调用，属于 Window 对象所有
    }
}

```

```

function g(){
    this.c = " c ";
    alert( this.a + this.b + this.c + this.d );    //返回 a b c undefined
    h();                                          //g 函数域内调用，属于 Window 对象所有
    function h(){
        this.d = " d ";
        alert( this.a + this.b + this.c + this.d ); //返回 a b c d
    }
}
f();                                          //在全局作用域内调用函数 f，属于 Window 对象所有

```

上面示例很有意思地演示了函数作用域与函数调用对象的关系。虽然说多层嵌套的函数结构中无法相互访问，但是它们的所有者都是相同的，即为 Window 对象。因此，当函数被调用后，变量 a、b、c 和 d 都属于全局对象 Window，因此都可以通过 Window 获取它们的值。

但是对于上面的函数嵌套结构，如果不通过函数调用的方式来执行，而是通过对象实例化的方式来激活，则会发现 this 关键字所指向的对象完全不同了。代码如下：

```

function f(){
    this.a = " a ";                          //属于对象 f 所有
    alert( this.a + this.b + this.c + this.d ); //返回 a undefined undefined undefined
    var x = new e();                          //实例化对象 e
    function e(){
        this.b = " b ";                      //属于对象 e 所有
        alert( this.a + this.b + this.c + this.d ); //返回 undefined b undefined undefined
        var x = new g();                      //实例化对象 g
        function g(){
            this.c = " c ";                  //属于对象 g 所有
            alert( this.a + this.b + this.c + this.d ); //返回 NaN c undefined
            var x = new h();                  //实例化对象 h
            function h(){
                this.d = " d ";              //属于对象 h 所有
                alert( this.a + this.b + this.c + this.d ); //返回 NaN d
            }
        }
    }
}
var x = new f();                            //实例化对象 f

```

在上面示例中，通过运算符 new 实例化函数对象，此时整个嵌套函数的结构性质就发生了变化，嵌套结构实际上就是由多个嵌套的对象组成，因此 this 关键字就在函数作用域内分别指向所在的函数对象，返回值的也就不同。如果在当前对象内没有直接定义的属性，则会返回 undefined，而当 undefined+undefined 时就会返回 NaN。

如果把函数的属性和方法指向 this，问题就复杂了。下面看一个示例：

```

function f(){
    this.x=function(x){                      //使用 this 关键字为函数定义一个方法 x()
        return x
    }
}
f();                                          //调用函数
alert(f.x(4));                              //使用函数 f 名调用方法 x()，则返回编译错误

```

虽然方法 x() 是在函数 f 体内定义，但是它已经不属于 f 函数了。此时由于所有者的原因，它应该属于

Window 对象。如果使用如下代码则可以正确调用方法 x():

```
alert(window.x(4)); //使用 window 对象调用方法 x(), 则返回 4
```

当然如果使用 this 来调用方法 x(), 则也是允许的, 因为此时的 this 正好指代 Window 对象:

```
alert(this.x(4)); //使用 this 调用方法 x(), 则返回 4
```

不过, 如果借助 new 运算符来实例化函数 f, 则此时函数 f 就变成了一个对象类, 实例化后的对象 a 就成为 this 关键字的所有者, 此时调用方法 x() 就不能够使用 Window 对象。例如:

```
function f(){
    this.x=function(x){ //使用 this 关键字为函数定义一个方法 x()
        return x
    }
}
```

//使用 new 运算符实例化函数 f, 并把实例化对象赋值给变量 a, 则 a 就拥有函数 f 的结构

```
var a = new f();
```

```
alert(a.x(4)); //使用对象 a 调用方法 x(), 则返回 4
```

此时如果使用 Window 对象来调用方法 x(), 则会显示编译错误。因为此时的 this 就不再指代对象 Window, 而是对象 a。例如:

```
alert(window.x(4)); //提示编译错误
```

当然使用 this 也是不允许的, 因为在全局作用域内, this 指代 Window 对象, 而不是对象 a。例如:

```
alert(this.x(4)); //提示编译错误
```

16.5 动态调用

动态调用函数一般使用 Function 对象定义的 call() 和 apply() 方法。call() 和 apply() 方法本质上是将特定函数当作一个方法绑定到指定对象上并进行调用。它们的语法格式如下:

```
function.call(thisobj, args...)
```

```
function.apply(thisobj, args)
```

其中参数 thisobj 表示指定的对象, 而 args 表示要传递给函数的参数。当函数被绑定到对象上之后, 将利用传递的参数执行函数, 并返回函数的值。例如:

```
function f(x,y){ //定义一个简单的函数
    return x+y;
}
function o(a,b){ //定义一个函数结构的伪对象
    return a*b;
}
alert(f.call(o,3,4)); //返回 7
```

在上面示例中, f 是一个简单的函数, 而 o 是一个对象。现在通过 call() 方法把函数 f 绑定到对象 o 身上, 变为它的一个方法, 然后动态调用该方法, 最后返回函数运行的值。实际上, 上面示例可以转换为下面代码:

```
function f(x,y){ //定义一个简单的函数
    return x+y;
}
function o(a,b){ //定义一个函数结构的伪对象
    return a*b;
}
o.m = f; //为对象 o 定义一个方法 m, 该方法将调用函数 f
alert(o.m(3,4)); //返回 7。调用对象 o 的方法 m
delete o.m; //删除对象 o 的方法 m
```

apply() 方法与 call() 方法没有太大区别, 只不过它们传递给函数的参数方式不同。其中, apply() 是以数

组形式进行参数传递，而 `call()` 方法可以同时传递多个值。另外，JavaScript 在 1.2 版本时就支持了 `apply()` 方法，而在 1.5 版本时才开始支持 `call()` 方法。例如，针对上面示例，使用 `apply()` 方法来实现。代码如下：

```
function f(x,y){
    return x+y;
}
function o(a,b){
    return a*b;
}
alert(f.apply(o,[3,4]));           //返回 7
```

当然这种用法上的差异却能够给具体应用带来很大的便利。例如，如果某个函数仅能够接收多个参数列表，而现在希望把一个数组的所有元素作为参数进行传递，此时使用 `apply()` 方法就显得非常便利。代码如下：

```
function max(){
    var m = Number.NEGATIVE_INFINITY; //声明一个负无穷大的数值
    for( var i = 0; i < arguments.length; i ++ ){ //遍历函数所有的实参
        if( arguments[i] > m ) //如果实参值大于变量 m，则把该实参值赋值给 m
            m = arguments[i];
    }
    return m; //返回最大值
}
var a = [23, 45, 2, 46, 62, 45, 56, 63]; //声明并初始化数组
var m = max.apply( Object, a ); //把函数 max 绑定为 Object 对象的方法，并动态调用
alert( m ); //返回 63
```

在上面示例中，首先定义了一个函数，用来计算所传递实参的最大值。由于该函数仅能够接收多个数值参数，现在通过 `apply()` 方法动态调用 `max` 函数，然后把它绑定为 `Object` 对象的一个方法，并借机把一个数组传递给它，最后返回函数的运行值。如果没有 `apply()` 方法，想使用 `max()` 函数来计算数组中最大元素值，就需要把数组所有元素全部读取出来，然后再传递给函数，显然这种做法是费力不讨好的。

实际上，也可以把数组元素通过 `apply()` 方法传递给系统对象 `Math` 的 `max()` 方法来计算数组的最大元素值。例如：

```
var a = [23, 45, 2, 46, 62, 45, 56, 63]; //声明并初始化数组
var m = Math.max.apply( Object, a ); //调用系统函数 max
alert( m ); //返回 63
```

使用 `call()` 和 `apply()` 方法时，应该注意以下几个问题：

- ☑ 使用 `call()` 和 `apply()` 方法可以把一个函数转换为方法传递给某个对象，但这种行为只是临时的，函数最终并没有作为对象的方法而存在。当函数被调用之后，该对象方法会自动被注销。下面示例很具体地说明了这种行为。

```
function f(){ //定义空函数
f.call( Object ); //把函数 f 绑定为 Object 对象的方法
Object.f(); //调用该方法，则返回编译错误
```

- ☑ 使用 `call()` 和 `apply()` 方法的目的是不为对象绑定方法，而是运行函数的一种技巧。绑定对象的方法仅是一种桥梁，而最终目的是运行函数。通过 `call()` 和 `apply()` 方法动态调用函数可以实现执行完毕即注销函数，避免资源占用，同时可以实现更灵活的调用。

使用 `call()` 和 `apply()` 方法动态调用函数，还能够实现调用一个对象的一个方法，以另一个对象替换当前对象，也就是说，更改对象的内部指针（`this` 指向对象），这在面向对象的编程过程中是非常有用的。例如：

```
var x = "o"; //定义全局变量 x，初始化为字符 o
function a(){ //定义函数类结构 a
    this.x = "a"; //定义函数内局部变量 x，初始化为字符 a
}
```

```

function b(){                //定义函数类结构 b
    this.x = "b";           //定义函数内局部变量 x，初始化为字符 b
}
function c(){                //定义普通函数，提示变量 x 的值
    alert( x );
}
function f(){                //定义普通函数，提示当前指针所包含的变量 x 的值
    alert( this.x );
}
f();//返回字符 o，即全局变量 x 的值。this 此时指向 Window 对象
f.call( window );           //返回字符 o，即全局变量 x 的值。this 此时指向 Window 对象
f.call( new a() );           //返回字符 a，即函数 a 内部的局部变量 x 的值。this 此时指向函数 a
f.call( new b() );           //返回字符 b，即函数 b 内部的局部变量 x 的值。this 此时指向函数 b
f.call( c ); //返回 undefined，即函数 c 内部的局部变量 x 的值，但是该函数并没有定义 x 变量，所以返回没有定义。
this 此时指向函数 c

```

通过上面示例的比较发现，函数 f 内部的 this 关键字会随着所绑定的对象不同而指向不同的对象。因此，利用 call() 或 apply() 方法能够改变函数内部指针指向所绑定的对象，从而实现属性或方法继承。

在函数体内，call() 和 apply() 方法的第 1 个参数都是关键字 this 的值。例如：

```

function f(){                //定义函数类结构
    this.a="a";              //定义成员 a 并赋值，a 为属性
    this.b = function(){     //定义成员 b 并赋值，b 为方法
        alert("b");
    }
}
function e(){                //定义函数
    f.call(this);            //在函数体内动态调用函数 f，this 指代函数 e
    alert(a);                //显示变量 a 的值
}
e()                          //返回字符串 a

```

如果在函数体内，使用 call() 和 apply() 方法动态调用外部函数，并把 call() 和 apply() 方法的第 1 个参数值设置为关键字 this，则当前函数 e 将继承函数 f 的所有成员。因此，使用 call() 和 apply() 方法能够复制调用函数的内部变量给当前函数体，就是更改了函数 f 内部关键字 this 指向了函数 e，这样函数 e 就可以引用函数 f 的内部成员。this 巧妙地改变函数的作用域，从而实现变量复制的目的。如果把其中的 f.call(this) 修改为 f.call(e) 或者 f.call(Object) 等，则无法实现这种偷梁换柱的目的。

在下面示例中将演示如何使用 apply() 方法循环更改当前指针，从而实现循环更改函数的结构。

```

function r( x ){              //定义一个简单的函数
    return ( x );
}
//定义一个稍复杂的函数，该函数将修改第 1 个参数值，并返回参数集合
function f( x ){
    x[0] = x[0] + ">";
    return x;
}
function o(){                //循环更改函数 r 中返回值
    var temp = r;
    r = function(){
        return temp.apply( this, f( arguments ) );
    }
}
function a(){                //定义函数 a

```

```

o();           //调用函数 o，修改函数 r 的结构，即返回值
alert( r( "=" ) ); //显示函数 r 的返回值
}
for( var i = 0 ; i < 10; i++ ){ //循环调用函数 a
    a();
}

```

执行上面示例，会看到提示信息框中的提示信息不断变化，如图 16.2 所示。该示例的核心就在于函数 o 的设计。在这个函数中，首先使用一个临时变量存储函数 r。然后修改函数 r 的结构，在修改的 r 函数结构中，通过调用 apply() 方法修改原来函数 r 的指针指向当前对象，同时执行原函数 r，并把执行函数 f 的值传递给它，从而实现修改函数 r 的 return 语句的后半部分信息，即为返回值增加一个前缀字符=。这样每次调用函数 o 时，都会为其增加一个前缀字符=，从而形成一种动态的变化效果。



图 16.2 apply() 方法应用示例效果

call() 和 apply() 方法应用是非常灵活的，在大型 JavaScript 技术框架中经常会使用它们，可以利用它们来实现动态更改对象的指针，从而实现各种复杂的功能。

16.6 函数作用域

JavaScript 执行环境就相当于一个操作平台。但是，对于 JavaScript 来说，执行环境是一个抽象的概念，它表示实现必要行为的域。

网页内包含的任何 JavaScript 代码都会在浏览器窗口这个平台运行（或称为全局执行环境，window）。但是对于函数来说，当函数被调用时，也会有一个局部的执行环境。

当调用函数时，系统会自动为其创建一个相对封闭的运行环境。如果调用另一个函数，则会创建另一个新执行环境。因此，在一个嵌套结构的函数结构内，会嵌套几个局部执行环境。

注意，这个执行环境是在调用函数时创建的，并不是在函数被解析时就会自动创建。同样可以推理，在递归调用函数时，会创建一个多层嵌套的执行环境。

函数总是在自己的执行环境中运行，如读写局部变量、函数参数、运行内部逻辑结构等。当然，对于函数来说，它可以跨越当前环境向上一级环境索取标识符（如变量、参数、属性等），这就是所谓的作用域链（请参阅下面小节的讲解）。当函数返回时（即函数运行完毕），会返回原始执行环境（即上一级执行环境）。这种运行过程实际上就是 JavaScript 构建的执行环境栈。

在创建执行环境的过程中，JavaScript 会遵循一定的运行规则，并按照代码顺序完成一系列操作。这个操作过程如下：

- （1）根据调用时传递的参数创建调用对象。
- （2）创建参数对象，存储参数变量。
- （3）创建对象属性，存储函数定义的局部变量。
- （4）把调用对象放在作用域链的头部，以便检索。
- （5）执行函数结构体内语句。
- （6）返回函数返回值。

针对上面的操作过程，详细描述如下。

首先，在函数执行环境中创建一个调用对象。调用对象与执行环境是两个不同的概念，也是另一种运行机制。提及对象，大家可以联系第 15 章对对象技术的深入剖析，它可以定义和访问自己的属性或方法。不过，这里的对象不是完整意思上的对象，它没有原型，且不能够被引用。这与 Arguments 对象的 arguments[]

数组一样，不是真正意思上的数组。

调用对象会根据传递的参数，创建自己的 `Arguments` 对象，这是一个结构类似数组的对象，该对象内部存储着调用函数时所传递的参数。接着创建名为 `arguments` 的属性，该属性引用刚创建的 `Arguments` 对象。

然后，为执行环境分配作用域。作用域由对象列表或对象链组成。每个函数对象都有一个内部属性（`scope`），这个属性值也是由对象列表或对象链组成的。`scope` 属性值构成了函数调用执行环境的作用域，同时，调用对象被添加到作用域链的头部，即该对象列表的顶部（即作用域链的前端）。

实际上，这个头部是针对该函数的作用域链而言的，也就是说，把调用对象排在函数作用域链的最上面。例如，在下面这个示例中，当调用函数 `e()` 时，将创建函数 `e()` 的调用对象，创建函数 `e()` 的作用域。但是在调用函数 `e()` 之前，会先调用函数 `g()`，并生成调用函数 `g()` 的对象。而调用函数 `e()` 的对象会在函数 `e()` 的作用域范围内处于头部位置，即排在最前面。代码如下：

```
function f(){           //定义普通函数
    return e();         //调用函数 e(), 并返回值
    function e(){       //嵌套函数 e()
        return g();     //调用函数 g(), 并返回值
        function g(){   //嵌套函数 g()
            return 1;    //返回值 1
        }
    }
}
alert(f());             //调用函数 f, 返回值 1
```

接着，就是正式执行函数体内代码了，此时 JavaScript 会对函数体内创建的变量执行变量实例化操作（即转换为调用对象的属性）。

将函数的形参也创建为调用对象的命名属性。如果调用函数时传递的参数与形参一致，则将相应参数的值赋给这些命名属性，否则将会给命名属性赋值为 `undefined`。

对于内部定义函数（请注意与嵌套函数的区分，两者语义不完全重合），会以其声明时所用名称为调用对象创建同名属性，对应的函数则被创建为函数对象，并被赋值给该属性。

将在函数内部声明的所有局部变量创建为调用对象的命名属性。请注意，在执行函数体内的代码，并计算相应的赋值表达式之前不会对局部变量执行真正的实例化。

由于 `arguments` 属性与函数局部变量对应的命名属性都属于同一个调用对象，因此可以将 `arguments` 作为函数的局部变量来看待。

最后创建 `this` 对象，并对其进行赋值。如果赋值是一个对象，则 `this` 将指向该对象引用。如果赋值是 `null`，则 `this` 就指向全局对象。

创建全局执行环境的过程与上面的描述稍微不同，因为全局执行环境没有参数，所以不需要通过定义调用对象来引用这些参数。但是全局执行环境会有一个作用域，即全局作用域，它的作用域链实际上只由一个对象组成，即全局对象（`Window`）。全局执行环境也会有变量实例化的过程，它的内部函数就是涉及大部分 JavaScript 代码的、常规的顶级函数声明。全局执行环境也会使用 `this` 对象来引用全局对象。

16.6.1 词法作用域与执行作用域

作用域（`scope`）是 JavaScript 语言的基石之一。JavaScript 作用域可以细分为词法作用域和动态作用域。

- ☑ 词法作用域：是从静态角度来说的。在函数没有被调用之前，根据函数结构的嵌套关系来确定函数的作用域。因此它取决于源代码，所以通常编译器可以进行静态分析来确定每个标识符实际的引用。
- ☑ 动态作用域：是从动态角度来说的。当函数被调用之后，它的作用域会因为调用而发生变化，此时作用域链也会随之进行调整。

也可以把 JavaScript 作用域分为定义作用域和执行作用域。定义作用域就是词法作用域，即说明函数在定义时存在的嵌套关系。但是当函数被执行时，作用域可能会发生变化，于是执行作用域也是动态作用域。JavaScript 函数运行在它们被定义的作用域里，而不是它们被执行的作用域里。

在早期的 JavaScript（即 1.2 以前）版本中，用户只能在全局作用域中定义函数，不允许函数嵌套，这时函数执行都是在全局作用域中，作用域链永远都是从函数局部作用域到 Window 全局作用域，即函数的调用对象都链接在全局作用域上。

在后来版本中，JavaScript 允许函数嵌套，于是作用域就变得非常复杂了。函数的词法作用域通过静态的代码结构就能辨析出来，但是它们的动态作用域就必须根据调用关系来动态确定。例如：

```
function f(){           //函数 f
    return e();         //执行函数 e，并返回运算值
    function e(){       //嵌套函数 e
        var a = 3;      //定义局部变量
        return a;       //返回局部变量的值
    }
}
f();                   //返回 3。调用函数 f
```

在上面的示例中，函数 f() 包含函数 e()，函数 e() 在函数 f() 中被执行，它们的词法作用域和动态作用域是重合的。作用域链包括函数 e() 的调用对象（自身）、函数 f() 的调用对象（自身）和全局对象（Window）。

```
function f(a){          //定义普通函数 f
    return function e(){ //返回一个匿名函数，匿名函数返回函数 f 的参数
        return a;
    }
}
var a = f(1);           //把匿名函数赋值给变量 a，则调用对象为 a
var b = f(2);           //把匿名函数赋值给变量 b，则调用对象为 b
var c = f(3);           //把匿名函数赋值给变量 c，则调用对象为 c
alert(a());             //返回 1
alert(b());             //返回 2
alert(c());             //返回 3
```

在上面的示例中，匿名函数的词法作用域是固定的，它的作用域链是匿名函数、函数 f() 和全局对象。但是，当在全局环境中调用函数 f，问题就变得很复杂了：

- ☑ 对于 a = f(1) 来说，函数 f() 的调用对象为 a，它的作用域链就是匿名函数、a 和全局对象。
- ☑ 对于 b = f(2) 来说，函数 f() 的调用对象为 b，它的作用域链就是匿名函数、b 和全局对象。
- ☑ 对于 c = f(3) 来说，函数 f() 的调用对象为 c，它的作用域链就是匿名函数、c 和全局对象。

虽然 3 个变量调用同一个函数，函数的结构也相同，但是由于作用域发生了变化，则返回值也各不相同。如果使用图示来表示，如图 16.3 所示。

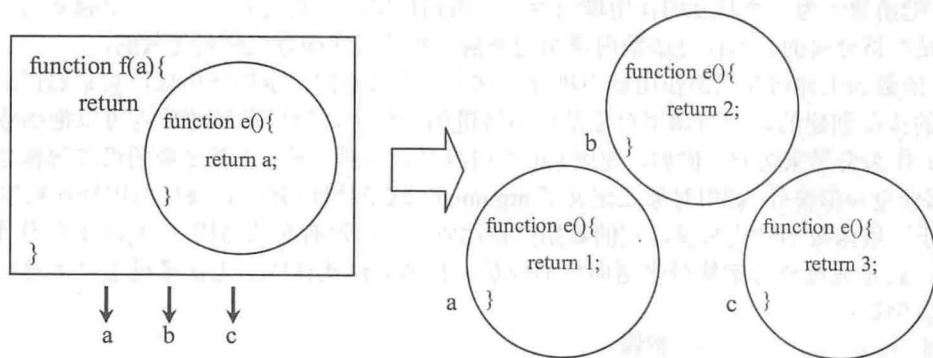


图 16.3 动态作用域的变化示意图

16.6.2 作用域链

作用域链是 JavaScript 解决变量访问的机制。JavaScript 规定每一个执行环境（作用域）都有一个与之相关联的作用域链。执行环境就是运行上下文（Execution Context），即代码运行环境。JavaScript 代码都是在特定运行上下文中执行的，当执行一个函数时，引擎会自动创建一个运行上下文环境，并在作用域链中添加这个作用域。当 JavaScript 解释器解释代码时，会根据这个作用域链，从局部向全局逐级检索，在特定的上下文中解析标识符。

对于 JavaScript 程序来说，每个运行环境（如浏览器中的页面）都会有一个与 JavaScript 执行环境相关的作用域链，这个作用域链就是一个对象列表。由于对象之间有着层级关系，串在一起犹如一串珍珠，因此可以说是对象链，习惯称之为作用域链。

当 JavaScript 代码需要存取变量的值时，JavaScript 解释器会就近查询当前作用域对应的对象是否存在同名属性。如果该对象有同名属性，那么就采用这个属性的值。如果该对象没有同名属性，则 JavaScript 就会向上继续查询作用域链中的上一级对象。如果上一级对象仍然没有同名属性，那么就继续向上查询对象，依此类推。最后，查询到作用域链的顶部（即全局对象），如果在全局对象中仍然没有找到同名属性，则返回 undefined 的属性值。在使用变量时，应注意以下 3 个问题：

- ☑ 全局变量具有全局作用域。
- ☑ 在函数体内声明的变量具有局部作用域。
- ☑ 如果一个定义函数（注意，是定义函数，而不是调用或执行函数）被嵌套在另一个函数中，那么在嵌套的函数体内声明的变量就具有嵌套的局部作用域。

下面以一个示例进行说明：

```
var a = 1;           //全局变量
(function(){
    var b = 2;       //第 1 层局部变量
    (function(){
        var c = 3;   //第 2 层局部变量
        (function(){
            var d = 4; //第 3 层局部变量
            alert(a+b+c+d); //返回 10
        })()          //直接调用函数
    })()              //直接调用函数
})();                //直接调用函数
```

在这个示例中，JavaScript 解释器首先在最内层调用对象中查询属性 a、b、c 和 d，其中只找到了属性 d，并获得它的值（4）。然后沿着作用域链，在上一层调用对象中继续查找属性 a、b 和 c，其中找到了属性 c，获得它的值（3）。依此类推，直到匹配到所有需要的变量值为止，如图 16.4 所示。

JavaScript 把函数作为一个独立的作用域而存在。所谓作用域，其实就是一个完全独立的空间，这个空间与外界空间是严格分隔的。因此在函数内声明的变量、参数，在函数域外是无效的。

JavaScript 函数的主体将在局部作用域中执行，这个作用域不同于全局作用域，它是通过把调用对象添加到作用域链的头部创建的。因为调用对象是作用域链的一部分，所以在函数体内可以把函数内部的成员（即对象属性）作为变量来访问。例如，使用 var 语句声明的局部变量，以及函数的形参等都是函数的对象属性。除了局部变量和形参外，调用对象还定义了 arguments 这个特殊属性，该属性引用特殊对象 Arguments。

但是，对于一般函数的结构来说，它的数据结构比较特殊，没有对象结构（Object）的作用域链特性。所谓作用域链，就是通过点号运算符来指向内部成员。在函数结构体中，无法通过点号运算符来引用内部包含的子成员。例如：

```
function f(){        //函数体
    function e(){    //子函数
        function g(){ //孙函数
```

```

    return 3;
  }
}
var b = true;           //函数的变量成员
var c = function(){     //函数的变量成员
  return "c";
}
}
alert(f.e.g());          //抛出错误
alert(f.c());            //抛出错误
alert(f.b);              //抛出错误

```

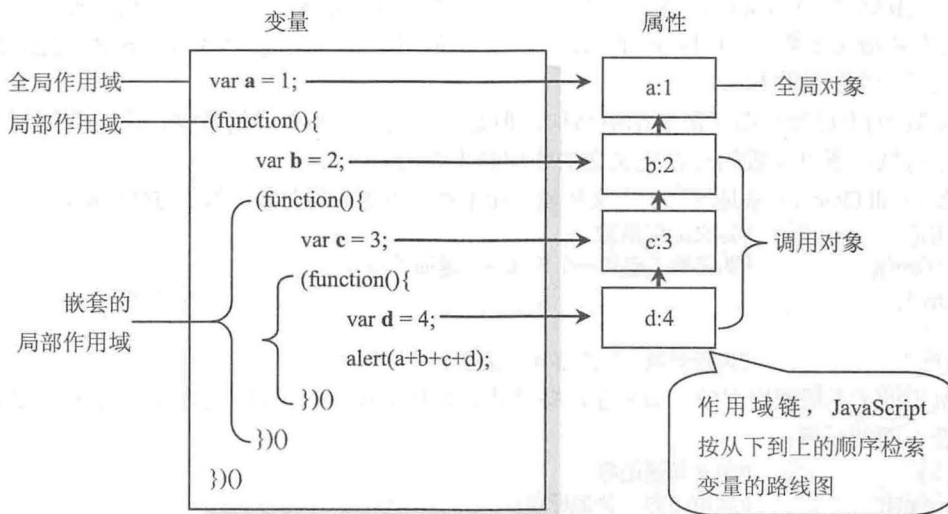


图 16.4 作用域链

在上面示例中, 函数 `f` 内部的结构是符合语法规范的, 但是无法通过点号运算符来引用它的成员。如果在对象内部是完全可以进行引用的。不过可以通过 `return` 语句返回函数内部变量, 以方便外界访问。例如, 在下面示例中可以调用成员函数 `g`:

```

function f(){
  return e;
  function e(){
    return g;
    function g(){
      return 3;
    }
  }
  var b = true;
  var c = function(){
    return "c";
  }
}
alert( f()()() );          //返回 3

```

16.6.3 调用对象

JavaScript 函数遵循词法作用域, 但不遵循执行作用域, 这意味着它们运行在自己定义的作用域中, 而

不是运行在执行它们的作用域中。

定义函数实际上就是定义作用域，即词法作用域，而不是定义动态的执行作用域。当然动态作用域不是定义的，也是无法静态确定的，它是根据调用对象不同而变化的。函数都有自己的作用域，且都会在自己的作用域内运行，而不是在调用它的动态作用域中执行。下面以 16.6.2 节的示例进行说明：

```
function f(a){
    return function e(){
        return a;
    }
}
var a = f(1);
```

当把函数 `f()` 赋值给变量 `a` 时，调用对象是 `a`，动态作用域也是 `a`，此时变量 `a` 内存储的是返回的匿名函数结构体，而不是指向函数 `f()` 的引用地址。此时变量 `a` 的结构就变成了匿名函数结构体。因此说，`var a = f(1)` 实际上是定义了一个函数作用域。

虽然在函数 `f()` 中已经声明了匿名函数结构，但是它仅是一个被传递的数据，在 `f` 作用域内，并没有定义匿名函数作用域，所以函数始终在定义它的作用域中执行。

调用对象 (Call Object) 就是运行上下文环境，而不是调用函数的对象。为了方便理解，下面举两个反例：

```
function f(){           //定义的空函数
f.a = function(){       //为函数 f 定义一个方法 a，返回数值 1
    return 1;
}
alert(f.a());           //调用对象 f 的方法 a，返回 1
```

上面示例中的 `f` 不是调用对象，如果在函数体内，使用 `this` 关键字进行引用，显示它仅是作用域链的上一级。继续看下面的示例：

```
function f(){           //定义普通函数
    return e();          //调用函数，并返回值
    function e(){        //嵌套函数 e
        return 1;        //返回值 1
    }
}
alert(f());             //调用函数 f，返回值 1
```

在上面的示例中，嵌套函数 `e` 在函数 `f` 中被调用，`f` 不是函数 `e` 的调用对象。当函数被调用时，系统会自动为它创建一个对象，它包含了函数参数属性（即 `arguments` 对象）。然后根据函数结构体内定义的各个局部变量名，在调用对象中定义相应的变量作为调用对象的属性，用来存储函数局部变量的数据。最后，将这个调用对象放在作用域链的头部。

16.7 闭包函数

闭包 (Closure) 是具有闭合特性的作用域，遵循特别作用域规则且可以用作参数的代码块。在 JavaScript 中，函数闭包是具有闭合作用域的匿名函数。闭包虽然结构与函数相同，但是它不全等于函数，因为还有一类对象闭包 (Object Closure)，对象闭包通过 `with` 语句来实现。例如：

```
with(o = {}){           //with 语句定义的对象闭包
    var a = function(){  //对象闭包的成员
        return "a";
    };
}
alert(a());             //直接读取闭包内成员。返回字符 a
```

上面的示例使用 `with` 语句定义了一个对象作用域，在这个封闭的域中包含了一个成员方法，但是 `with` 语句所定义的对象闭包却允许外界访问它的成员。使用函数作用域来表示相同的结构。代码如下：

```
function f(){
    var a = function(){           //匿名函数定义的函数闭包
        return "a";
    };
    return a;
}
alert(f());                       //调用闭包结构，返回字符 a
```

如果把函数闭包和对象闭包混合在一起使用，则它们将表示为不同的作用域，并实现相互通信。例如：

```
function Me(){                   //定义函数对象
    this.name = "css8";          //定义对象属性
    this.saying = function(){    //定义对象方法
        return "How are you?";
    }
}
var me = new Me();               //实例化对象
with( me ){                     //定义 me 实例作用域
    alert(name + "：" + saying()); //访问作用域内的成员
}
```

16.7.1 认识闭包

函数式编程的基本技巧就是把大量的逻辑封装在表达式中来完成，因此整个程序犹如一串连续的运算。例如，下面是一个连续运算的表达式，该表达式实际上就是一个逻辑复杂的分支结构，并在分支结构中包含函数结构体，以判断两种表达式的大小，并输出提示信息。可以看到，整个代码是在无命令语句的情况下完成任务的，与命令式语言风格迥然不同。

```
(( function f( x, y ){
    return ( x + y ) * ( x + y );
} )( 25, 36 ) >
( function f( x, y ){
    return x * x + y * y;
} )( 25, 36 ) ) ?
alert( "( x + y )^2" ) : alert( "x ^2+ y^2" ) //返回提示信息 ( x + y )^2
```

在 JavaScript 函数式编程中，函数被大量地使用，但是如何解决下面两个问题：

- ☑ 数据存储和传递问题。由于程序是在大量的表达式中一口气运算实现的，其中生成的数据该如何寄存，又如何传递？如果说，函数式编程无法解决数据寄存与传递问题，整个设计模式也就无从实现。
- ☑ 数据安全与相互干扰问题。大量的数据在同一个表达式中被执行，如何确保数据之间的安全，而不会出现数据之间相互干扰，同时又能够保证数据之间可以有序交流。

闭包函数能够很好地解决上面两个问题，以确保连续运算在复杂环境中下能够正确执行。闭包是 JavaScript 语言核心特性之一，闭包的创建非常容易，有时候会在无意识中创建多个闭包函数。

简单描述，闭包就是内部函数（也可以称之为嵌套函数结构）。也就是说，在一个函数内定义的一个函数或函数表达式。从本质上来分析，实际上任何函数都是闭包，即使是在全局作用域中定义的函数，它也是全局对象 `Window` 的闭包，只不过我们很少这样使用，或者说它已经成为我们的行为习惯。

作为闭包的必要条件，内部函数还应该可以访问外部函数中声明的所有局部变量、参数和声明的其他内部函数。

当上述的两个必要条件实现后，此时如果在外部函数外调用这个内部函数，它就成为了闭包函数。简单地说，就是嵌套的内部函数在外部函数返回后（即被调用后）被执行。

当这个内部函数被执行时，它仍然需要访问外部函数中的局部变量、参数以及其他内部函数。虽然，这时外部函数已经被注销，外部函数的局部变量、参数和函数声明（最初时）也会随之被丢弃。但是，那些被内部函数引用的标识符所存储的值却被保留下来，并能够在生命期内与内部函数保持联系。

下面继续以这个经典的闭包示例为基础来演示上述抽象的道理：

```

1 function f(x){           //外部函数
2     var a = x;           //外部函数的局部变量，并把参数值传递给它
3     var b = function(){  //内部函数
4         return a;        //访问外部函数中的局部变量
5     };
6     a++                  //访问后，动态更新外部函数的变量
7     return b;            //返回内部函数
8 }
9 var c = f(5);            //调用外部函数，并赋值
10 alert(c());              //调用内部函数，返回外部函数更新后的值 6

```

演示步骤说明如下：

（1）程序预编译之后，程序从第 9 行开始解析执行，创建执行环境，创建调用对象，把参数和局部变量、内部的函数转换为对象属性。

（2）执行函数体内代码。在第 6 行执行局部变量 a 的递加运算，并把这个值传递给对象属性 a，同时内部函数动态保持与局部变量 a 的联系，也更新自己内部调用变量的值。

（3）外部函数把内部函数返回给全局变量 c，实现内部函数的定义，此时 c 完全继承了内部函数的所有结构和数据。

（4）外部函数返回后（即返回值后，也即调用完毕）会自动销毁，内部的结构、标识符和数据也随之丢失。

（5）执行第 10 行代码命令，调用内部函数，此时返回的是外部函数返回时（销毁之前）保存的变量 a 所存储的最新数据值，即返回 6。

如果没有闭包函数的作用，那么这种数据寄存和传递就无法得以实施。例如：

```

1 function f(x){
2     var a = x;
3     var b = a;           //直接把局部变量的值传递给局部变量 b
4     a++
5     return b;            //返回局部变量 b
6 }
7 var c = f(5);
8 alert(c);                //返回值为 5

```

通过上面的示例，可以很直观地看到，在没有闭包函数的辅助下，第 8 行代码执行后返回的值并没有与外部函数的局部变量 a 最后更新的值保持一致。也许把第 3 行代码移至到第 4 行代码之后，能够实现相同的目的。但是在实际开发中，程序的复杂性会远远超过上面这个示例，此时如果没有闭包的赋值，不仅不能够实现数据的即时更新，更为重要的是数据寄存也是面临的一个难题。

16.7.2 闭包基本特性

闭包结构有下面两个比较鲜明的特性。

1. 自闭特性

自闭特性表现为外界无法访问闭包结构内部的数据，如果在闭包内声明变量，外界是无法访问的，除

非闭包主动向外界提供访问接口，而这与函数结构具有惊人的一致性。例如：

```
function f(){
    var a = 1;
}
```

上面示例是一个非常简单的函数结构，其内部定义了一个变量 `a`。在全局作用域内，任何对象都无法访问变量 `a` 的值，此时函数 `f` 就具有自闭的特性，因此可以把函数 `f` 看做是 Window 对象的一个闭包体。但是闭包对外界有影响力，如调用外部的变量或属性，向外界返回值等。

2. 包裹特性

包裹特性表现为闭包具有保护特征。对于一般函数来说，当调用完毕之后，系统会自动注销函数。而对于闭包结构来说，当外部函数被调用之后，闭包结构依然被保存在系统中，闭包中的数据依然完好无损地保存着，从而实现包裹数据的目的。例如：

```
function f( x ){           //定义普通函数 f()
    var a = x;             //把参数存储在函数内局部变量中
    var b = function(){    //定义一个闭包，并赋值给局部变量 b
        return x;          //返回函数参数 x
    }
    return b;              //返回闭包结构
}
var c = f( 1 );            //调用函数，并传递给它一个值
alert(c());                //返回 1。调用闭包函数
```

上面示例中，首先在函数 `f` 结构体内定义两个变量，分别存储参数和闭包结构，而闭包结构中寄存着参数值。当调用函数 `f` 之后，函数结构被注销，随之它的局部变量也被注销，因此变量 `a` 中存储的参数值也随之丢失。但是变量 `b` 存储着闭包结构，因此闭包结构内部的参数值并没有被释放，当函数调用之后，依然能够从闭包结构中读取到参数值。从上面示例也可以总结出，在嵌套结构的函数结构中，内部的函数结构实际上就是一个闭包结构。

16.7.3 闭包基本用法

初步了解了闭包的基本特性，下面通过几个示例来演示闭包的应用，以便能够更透彻地理解什么是闭包，以及闭包的作用和用法。

【示例 3】 使用闭包结构能够跟踪动态环境中数据的实时变化，并即时存储。

```
function f(){              //定义普通函数 f()
    var a = 1;             //定义局部变量 a，初始值为 1
    var b = function(){    //定义一个闭包，并赋值给局部变量 b
        return a;          //返回函数参数 x
    }
    a++;                   //动态更新函数内局部变量 a 的值
    return b;              //返回闭包结构
}
var c = f();               //调用函数
alert(c());                //返回 2。而不是返回 1
```

在上面示例中，闭包中的变量 `a`，其存储的值并不是从上面行变量 `a` 的值简单拷贝，而是继续引用外部函数定义的局部变量 `a` 中的值，直到外部函数 `f` 调用返回。

【示例 4】 闭包不会因为外部函数环境的注销而消失，并始终存在。

```
<script language="javascript" type="text/javascript">
function f(){              //定义普通函数 f()，包含多个闭包的外部环境
    var a = 1;             //定义函数内局部变量 a，并设置初始值为 1
```



```

    b = function(){                //闭包 b
        alert( a );                //寄存函数内局部变量 a 的值，并进行提示
    }
    c = function(){                //闭包 c
        a ++ ;                     //递增并寄存函数内局部变量 a 的值
    }
    d = function( x ){             //闭包 d
        a = x;                     //传递并寄存函数内局部变量 a 的值
    }
}
</script>
<button onclick="f()">按钮 1: (f(    ))()</button><br />
<button onclick="b()">按钮 2: (b = function(){alert( a );})();</button><br />
<button onclick="c()">按钮 3: (c = function(){a ++ ;})();</button><br />
<button onclick="d(100)">按钮 4: (d = function( x ){a = x; })
(100)</button><br />

```

在上面示例中，普通函数 f 中定义了 3 个闭包函数，它们分别指向并寄存局部变量 a 的值，并根据不同的操作动态跟踪变量 a 的值。

当在浏览器中预览时，首先应该单击“按钮 1”，调用函数 f，将生成 3 个闭包，同时 3 个闭包同时指向局部变量 a 的引用。因此当函数 f 返回时，3 个闭包函数都没有被注销，而变量 a 由于被闭包引用而继续存在。这时如果直接单击按钮 2~4，则由于没有在系统中生成闭包结构，会弹出编译错误。

单击“按钮 3”，则将动态递增变量 a 的值。此时如果单击“按钮 2”，则会弹出提示值为 2。如果单击“按钮 4”，则向变量 a 传递值 100，将动态改变闭包中寄存的值。此时如果单击“按钮 2”，则会弹出提示值为 100。

【示例 5】 本示例将介绍如何利用闭包存储变量所有变化的值。

```

function f( x ){                  //定义功能函数，把参数数组的元素以闭包体分别封装在数组中并返回
    var a = [];                  //定义临时数组
    for ( var i = 0; i < x.length; i ++ ){ //遍历参数数组
        var temp = x[i];         //临时存储每个数组元素的值
        a.push( function(){      //把数组元素值封装在闭包中投入到临时数组 a 中
            alert( temp + ' ' + x[i] ) //闭包中被封装的参数数组元素值
        });
    }
    return a;                    //返回临时数组 a
}
function e(){                    //定义普通函数
    var a = f( ["a", "b", "c"] ); //调用函数 f(), 并向其传递一个数组
    for ( var i = 0; i < a.length; i ++ ){ //遍历函数 f()返回的数组
        a[i]();                  //调用闭包，查看内部封装的数组元素的值
    }
}
e();                             //调用函数 e

```

在这个示例中，函数 f 的功能是：把数组类型的参数中每个元素的值分别封装在闭包结构中，然后把闭包存储在一个数组中，并返回这个数组。

但是，在函数 e 中调用函数 f，并向其传递一个数组 (["a", "b", "c"]), 然后遍历函数 f 返回数组，结果发现，数组中每个元素的值都是 c undefined。为什么不是预期所传递的数组元素的值？

原来闭包中的变量 temp 并不是固定的，它会随时根据函数运行环境中的变量 temp 的值变化而更新，导致临时数组元素的值都是字符 c，而不是 a、b、c。同时由于循环变量 i 递增之后，最后的值是 3，则 x[3] 超出了数组的长度，结果就是 undefined。

解决闭包存在的这个问题：可以为闭包再包裹一层函数，然后运行该函数，并把外界动态值传递给它。当这个函数接收这些值，然后传递给内部的闭包函数，自己就注销了，从而阻断了闭包与最外层函数的实时联系。具体代码如下：

```
function f( x ){
//定义功能函数，把参数数组的元素以闭包体分别封装在数组中并返回
    var a = [];
    for ( var i = 0; i < x.length; i ++ ){
        var temp = x[i];
        a.push(
            ( function( temp, i ){
                return function(){
                    alert( temp + ' ' + x[i] )
                }
            })( temp, i )
        );
    }
    return a;
}
function e(){
    var a = f( ["a", "b", "c"] );
    for ( var i = 0; i < a.length; i ++ ){
        a[i]();
    }
}
e();
```

//定义临时数组
//遍历参数数组
//临时存储每个数组元素的值
//把被阻断的闭包投入到临时数组 a 中
//运行函数，阻断内部的闭包与外部环境的联系
//返回闭包函数，该函数保存的值是静态的
//为运行函数传递外部动态值
//返回临时数组 a
//定义普通函数
//调用函数 f，并向其传递一个数组
//遍历函数 f 返回的数组
//调用闭包，查看内部封装的数组元素的值
//调用函数 e

【示例 6】 本示例将介绍如何利用同一个闭包体声明多个闭包。同一个闭包，通过分别引用，能够在当前环境中生成多个闭包。例如：

```
function f( x ){
    var temp = x;
    return function( x ){
        temp += x;
        alert( temp );
    }
}
var a = f( 50 );
var b = f( 100 );
a( 5 );
b( 10 );
```

//定义普通函数
//返回闭包
//生成第 1 个闭包
//生成第 2 个闭包
//返回 55
//返回 110

16.7.4 闭包标识系统

从结构上来说，闭包函数与普通函数没有什么两样，主要包含以下类型的标识符：

- ☒ 函数参数（形参变量）
- ☒ arguments 属性
- ☒ 局部变量
- ☒ 内部函数名
- ☒ this（指代闭包函数自身）

其中，this 和 arguments 是系统默认的函数标识符，不需要特别声明。这些标识符在闭包体内的优先级是（其中左侧优先级要大于右侧）this→局部变量→形参→arguments→函数名。

例如，下面的示例将在函数结构内显示函数结构的字符串：

```
function f(){           //定义函数
    alert(f)           //提示函数结构
}
```

f();//调用函数，返回函数 f 结构的字符串，等于 f.toString()

但是，如果在函数 f 中定义形参 f，则同名情况下参数变量的优先权会大于函数的优先权。例如：

```
function f(f){         //定义形参与函数同名
    alert(f)           //提示标识符 f 的值
}
f(true);              //返回 true，而不是函数 f 的结构字符串
```

继续比较形参与 arguments 属性的优先级：

```
function f(arguments){ //函数形参名与参数属性 arguments 同名
    alert(typeof arguments) //提示参数的类型
}
f(true);              //返回 boolean，而不是属性 arguments 的类型 object
```

上面示例说明了形参变量会优先于 arguments 属性对象。继续比较 arguments 属性与函数名的优先级：

```
function arguments(){  //定义函数名与 arguments 属性名同名
    alert(typeof arguments) //返回 arguments 的类型
}
arguments();           //返回 arguments 属性的类型 object，而不是函数的类型 function
```

上面示例的用法在 JavaScript 中会提示编译错误，因为系统不允许使用默认关键字来定义标识符的名称。

继续比较局部变量和形参变量的优先级，先看下面的示例：

```
function f(x){         //定义普通函数
    var x = 10;        //定义局部变量并赋值
    alert(x);          //显示变量 x 的值
}
f(5);                 //传递参数值为 5，返回提示为 10
```

上面的示例说明函数内局部变量要优先于形参变量的值。但是，如果局部变量没有赋值，则会选择形参变量。例如：

```
function f(x){         //定义普通函数
    var x;             //定义局部变量
    alert(x);          //显示变量 x 的值
}
f(5);                 //传递参数值为 5，返回提示为 5
```

因此，局部变量与形参变量重名时，如果局部变量没有赋值，则形参变量要优先于局部变量。下面这个示例很有意思，它说明了当局部变量与形参变量混在一起使用时，它们之间存在的微妙关系：

```
function f(x){
    var x = x;         //把形参 x 传递给局部变量 x
    alert(x);
}
f(5);                 //返回提示为 5
```

如果从局部变量与形参变量之间的优先级来看，则 var x = x 左右两侧都应该是局部变量。由于 x 初始化为 undefined，所以该表达式就表示把 undefined 传递给自身。但是从上面的示例来看，这说明左侧的是由 var 语句声明的局部变量，而右侧的是形参变量。也就是说，如果当局部变量没有初始化时，应用的是形参变量优先于局部变量。

16.7.5 闭包函数作用域

JavaScript 是一种动态语言，因此作用域和生命周期是该语言的一个重要特征。作用域决定了标识符的

可见性，即可见区域，它表示在指定范围内可用。从状态来看，作用域可以包括语法作用域和活动作用域。但是从使用的角度上分析，作用域主要包括表达式作用域、局部作用域和全局作用域 3 种类型。

对于 JavaScript 语言来说，一般遵循如下规则：

- ☑ 使用 `var` 语句（或不用）声明的全局变量，则在全局范围内有效（或称为可见）。
- ☑ 如果变量在任意位置隐式声明使用（即不使用 `var` 语句），则在全局范围内有效。
- ☑ 变量在函数体内显式声明（使用 `var` 语句声明），则仅在函数体内可见。
- ☑ 如果变量在函数体内有效，则在其所有内嵌函数中都有效。例如，下面的示例中参数变量 `x` 是函数 `f` 的私有变量，该变量将在函数 `f` 内部所有内嵌函数中都是可见的：

```
function f(x){
    return function(){
        return function(){
            return function(){
                return x;
            }
        }
    }
}
```

```
var a = f(1);           //调用函数
alert(a())()()         //返回 1
```

- ☑ 函数内声明的局部变量能够覆盖外部同名变量的值。例如，下面的示例中，内部函数的同名局部变量会逐层覆盖，并显示最里层的变量值：

```
function f(){           //普通函数
    var x = 1;           //局部变量
    return function(){
        var x = 2;       //覆盖上一级变量 x
        return function(){
            var x = 3;    //覆盖上一级变量 x
            return function(){
                return x;
            }
        }
    }
}
```

```
var a = f();           //调用函数
alert(a())()()         //返回 3，而不是 2 或 1
```

- ☑ 如果在内部函数中声明局部变量时，该作用域内所有引用外部同名变量值将被覆盖，初始化显示为 `undefined`。例如：

```
function f(x){
    return function(){
        var x;           //声明的局部变量，将覆盖外部同名参数变量
        return x;
    }
}
var a = f( 1 );         //调用函数
alert( a() )            //返回 undefined，而不是 1
```

通过语法分析可以看到，函数被解析时，会把内部的所有使用 `var` 语句声明的变量列入调用对象内的局部变量列表中，然后根据作用域链逐层上访变量。如果在当前作用域内发现了该变量，则会使用该变量，否则就会向上访问同名变量。如果变量为隐式使用，则将作为全局变量被列入全局作用域内的全局对象变量列表中。

16.7.6 闭包函数生存周期

变量生存周期表示变量从开始被分配内存地址到被销毁的过程。JavaScript 语言采用自动垃圾收集机制，但是规范中并没有详细说明垃圾回收的细节，而是根据具体实现来决定。不过 JavaScript 垃圾收集操作只赋予了很低的优先级。

垃圾回收的思路是这样的：如果对象不再可引用时，由于不存在对它的引用，使执行代码无法再访问到它，该对象就成为垃圾收集的目标。因而，在将来的某个时刻会将这个对象销毁并将它所占用的一切资源释放，以便操作系统重新利用。

在正常情况下，当退出一个执行环境时就会满足类似的条件。此时，作用域链结构中的调用对象，以及在该执行环境中创建的任何对象（包括函数对象）都不再可引用，因此将成为垃圾收集的目标。例如：

```
function f(x,y){           //外部函数
    var a = 1;             //声明和初始化局部变量
    function e(z){         //内部函数
        return x+y+z+a;
    }
    return e;              //返回内部函数
}
var b = f(2, 3);           //调用外部函数
```

在该示例中，调用外部函数 `f`，在执行环境中所创建的调用对象就不会被当作垃圾收集，因为该调用对象被一个全局变量 `b` 所引用，而且仍然是可以访问的，甚至可以通过 `b(n)` 来执行。同时，在这个被变量 `b` 引用的内部函数对象的作用域链中，包含属于创建该内部函数对象的执行环境的活动对象。

由于在执行被 `b` 引用的函数对象时，每次都要把该函数对象所引用的整个作用域链添加到内部函数创建的执行环境的作用域中，该作用域中包括内部执行环境的调用对象、外部执行环境的调用对象、全局对象。所以这个外部执行环境的调用对象也不会被当作垃圾收集。

由于调用对象受限于内部函数对象（现在被全局变量 `b` 引用）的作用域链引用，所以调用对象连同它的变量声明（即属性的值）都会被保留。而在对内部函数调用的执行环境中进行作用域解析时，将会把与调用对象的命名属性一致的标识符作为该对象的属性来解析。活动对象的这些属性值即使是在创建它的执行环境退出后，仍然可以被读取和设置。

在上面的例子中，当外部函数返回（退出它的执行环境）时，在其调用对象的变量声明中依然记录了形参、内部函数定义以及局部变量的值。`x` 属性值为 2，`y` 属性值为 3，局部变量 `a` 的值是 1，还有一个 `e` 属性，它引用由外部函数返回的内部函数对象。

生存周期只有两个：函数内部局部执行期间和函数外全局执行期间。对于闭包结构来说，主要依赖于函数实例被引用、释放引用和销毁的周期情况。例如：

```
function f( ){
    var a = 1;
    var e = function(){
        return a;
    }
    function g(x){
        var y = x + 1;
        e();
    }
    g(a);
}
```

在这个示例中，函数 `g()` 访问了两个存在依赖的变量。一是通过参数变量 `x` 引入了外部变量 `a`，该外部

变量 `a` 将在函数内被持有。由于参数仅作为函数内表达式的一部分参与运算，因此在闭包退出后，参数变量所绑定的外部变量 `a` 将被复位。因此，对于 `g(a)` 来说，函数被调用之后，即释放资源，从而说明外部变量 `a` 并没有被函数 `g()` 长期持有。

但是，对于变量 `e` 来说，由于该变量内闭包与外部变量 `a` 存在长期联系，从而导致如果闭包函数 `g()` 还存活时，外部函数 `f` 也将存在。这对于动态语言的 JavaScript 来说，外部函数内的局部变量 `a` 可能会随时发生变化，同时函数 `g()` 也可能被其他变量所引用，因此当函数 `f` 调用返回后，变量 `a` 也不会被清除。

除了在闭包体内通过标识符显式引用外部函数的变量值，从而导致闭包与闭包的生存周期发生关联以外，还有一种情况会导致闭包之间相互关联。例如：

```
function f(){
    var e = function(){
        return 1;
    }
    function g(x){
        var y = x;
        function h(){
            e();
        }
    }
    g(e);
}
```

在这个示例中，函数 `e()` 被函数 `g()` 中的局部变量 `y` 获得了一个引用。这与上面示例中传入的值数据，且参与表达式运算不同，它是对外部函数的引用，从而导致闭包与外部变量之间建立关联。

16.7.7 比较函数和闭包

在 JavaScript 中，函数仅是一段代码，可以把它理解为静态文本。在被调用之前，函数仅是词法意思上的结构，没有实际的价值。包括在 JavaScript 解释器预编译函数时，也仅是简单地分析函数的词法、语法结构，并根据函数标识符预定一个函数占据的内存空间，其内部结构和逻辑并没有被运行。

但是，一旦函数被调用执行，则此时闭包体也会随之诞生。可以这样说，闭包是函数运行期中的一个动态环境，它是一个动态概念，与函数的静态性是截然不同的概念。由于每个函数都是一个独立的上下文环境（即执行环境），因此当闭包函数被再次执行或者通过某种方法进入函数体时，就可以获取闭包内包含的信息。两者的简单比较如表 16.2 所示。

表 16.2 函数与闭包的比较

项 目	函 数	闭 包
功能	设计逻辑结构	存储和传输数据
状态	静态词法域（代码文本）	动态环境
时间	即时性（调用返回后即消失）	延迟性（在函数返回后，还能够存在）
作用域	根据词法作用域，可以事先确定作用域的关系	动态作用域，只有在具体的环境中调用函数时，才能够确定其作用域

闭包可以说是函数的数据包，用于存储数据。这个数据包在函数执行过程中处于激活状态。所谓激活状态，就是能够访问这个闭包及其存储的数据，当函数调用返回之后即被注销，但是闭包结构仍然存在，且保存着与函数关联的最终数据状态。

闭包中存储的数据包含函数运行实例的引用、环境表（即用来查找全局变量的表），以及由众多标识符构成的数组。在函数运行时，闭包可以实时访问上一级函数作用域中其他标识符的值。同一个函数中所有

闭包都可以引用函数体内相同标识符的值，而且相互影响。例如，下面的示例能够很好地说明在一个函数内多个闭包对函数所属标识符的影响，以及闭包之间的相互影响：

```
function f(){           //函数结构体
    var x = 5;           //函数局部变量
    function a(){        //闭包结构 a
        var y = x;       //把函数 f()的局部变量值递给闭包的局部变量 y
        alert(y);        //提示局部变量 y 的值
    }
    function b(){        //闭包结构 b
        x = x*x;         //动态更新函数 f()的局部变量 x
    }
    b();                 //调用闭包 b，则将更新函数 f()的局部变量 x 为 25
    a();                 //调用闭包 a，则显示函数 f()的局部变量 x 值变化 25
}
f();                    //调用函数 f()，创建运行环境
```

16.7.8 闭包函数与函数实例

函数实例就是对函数结构的克隆。函数实例与原函数是两个不同的实体对象，它们在内存中的位置是不同的，因此应该把函数实例与函数引用区分开来。闭包函数实际是与函数实例有着某种联系的。

例如，下面是一个简单的函数结构，具有返回值，但是使用 `new` 运算符实例化该函数之后，实例对象 `a` 却无法访问：

```
function f(){           //普通函数
    var x = 5;
    return x;           //函数返回值
}
var a = new f();        //实例化函数
alert(a.x)              //返回 undefined
alert(a.x)              //不是函数，无法访问
```

如果在函数体内使用点运算符为函数定义属性，则可以通过点运算符来访问函数成员变量：

```
function f(){           //函数对象
    this.x = 5;         //对象属性
}
var a = new f();        //实例化函数
alert(a.x)              //返回 5
```

也就是说，函数实例是基于函数对象的基础上才能够有效访问的，当然使用 `new` 运算符能够对任意函数进行实例化。也就是说，只有构造函数才能够执行有效实例化操作。

下面要分析的是函数实例与闭包函数的关系，也就是分析函数对象的实例是不是都包含闭包。先看一个示例：

```
function F(x){          //定义函数对象
    this.y = function(){ //定义对象方法
        return x;
    }
}
var a = new F(5);       //实例化函数
var b = new F(5);       //实例化函数
alert(a===b)            //返回 false，说明是不同的实例
alert(a.y===b.y)        //返回 false，说明是不同的实例
```

上面的示例是通过构造函数的方法创建一个函数对象 `F`，对象中包含一个方法 `y`。分别在变量 `a` 和 `b` 中

实例化对象 F，则发现它们的实例并不相同，而且它们的方法也是不同的，说明 a 和 b 属于不同的实例，且它们的方法也是分别独立的闭包结构体。

但是使用原型方法定义函数对象时，实例化对象及它们的方法存在不同的结果，这说明实例化的仅是对象结构本身。而对于对象的原型方法，则仅是引用，没有产生实例化对象方法，因此也就无从分析闭包的执行环境问题。如果用示意图来演示则如图 16.5 所示。

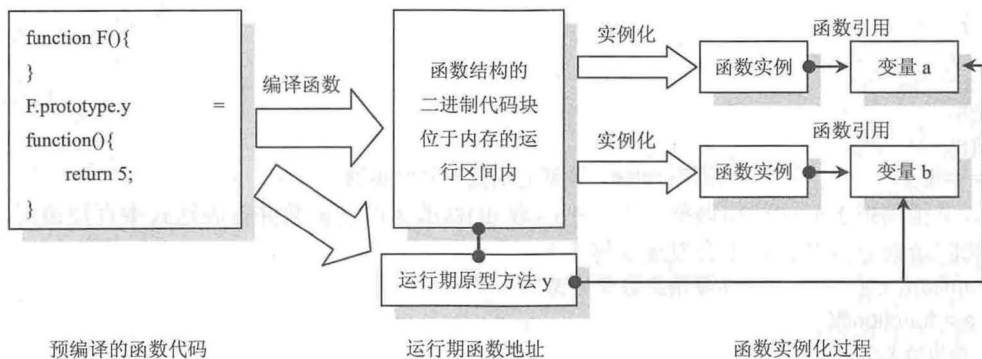


图 16.5 原型对象实例化过程示意图

使用不同的方法创建的函数对象，它们是否包含闭包结构还需要具体分析，这里主要观察函数对象被实例化之后，是否生成了实例化的方法。

```
function F(){           //定义函数对象
F.prototype.y = function(){ //定义对象的原型方法
    return 5;
}
var a = new F();         //实例化函数
var b = new F();         //实例化函数
alert(a===b)             //返回 false，说明是不同的实例
alert(a.y===b.y)         //返回 true，说明是对原型方法的引用
```

下面这个示例说明了 a 和 b 是两个不同的闭包结构。但是如果从本质来分析，它们实际上也是两个具体的函数实例，而不是对函数的引用。

```
function f(x){
    return function(){
        return x
    }
}
var a = f(5);
var b = f(5);
alert(a===b)           //返回 false，说明是不同的实例
```

因此，如果从函数实例的角度来分析闭包函数，会更容易理解。在下面这个示例中，a 和 b 看起来似乎是两个不同的实例，实际上它们都是对同一个函数 F() 的引用，因此这里也就没有闭包环境了。

```
function F(){           //定义一个空函数对象结构
F.y = function(){       //为对象定义具体的方法
    return 5;
}
var a = F;
var b = F;
alert(a===b)           //返回 true，说明是对函数的引用
alert(a.y===b.y)       //返回 true，说明是对函数的引用
```


下面这种多层嵌套的函数结构，函数 `f` 内部嵌套了两层函数。不过调用函数 `f()` 会返回一个双层的闭包结构，且变量 `a` 和 `b` 分别引用的是不同函数实例，因此它们都属于不同的闭包结构。

```
function f(x){           //普通函数
    return function(){   //返回匿名函数体
        return function(){ //返回匿名函数体
            return x;      //返回局部变量值
        }
    }
}
var a = f(5);
var b = f(5);
alert( a === b )         //返回 false，说明它们是不同的实例
```

但是，如果稍稍把上面的结构调整一下，把函数 `f()` 修改为匿名函数并在表达式中直接调用，同时把内部的多层嵌套的函数分拆开来，则会发现 `a` 等于 `b`。

```
var f = function(x){     //复杂函数表达式
    var e = function(){
        return x;
    }
    return function(){
        return e;
    }
}(5);
var a = f();              //调用表达式的返回实例
var b = f();              //调用表达式的返回实例
alert( a === b )         //返回 true，说明它们引用相同的函数，即实例函数
```

首先，变量 `f` 存储着一个调用的函数表达式，编译之后自动生成一个匿名函数结构体，同时在函数结构体内又定义了一个匿名函数结构，此时可以称之为函数实例或闭包，并把它指向局部变量 `e`。

然后，当这个匿名函数结构体被调用时，将返回的是变量 `e` 存储的值，该值指向已经定义的匿名函数的入口指针地址，而不是匿名函数结构体本身。因此，在变量 `a` 和变量 `b` 中，我们看到的是相同的闭包结构的引用地址值，如图 16.6 所示。

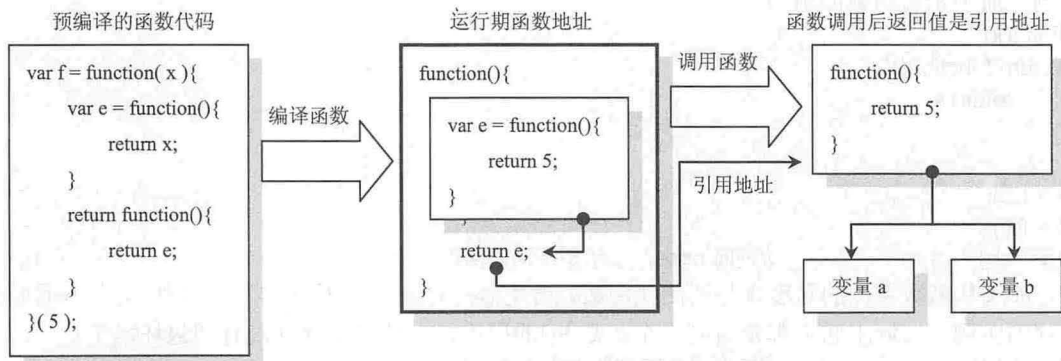


图 16.6 多层匿名函数嵌套过程示意图

调整一下上面示例的结构：

```
function f(x){           //普通函数
    var e = function(){   //内部匿名函数
        return x;
    }
    return function(){    //返回嵌套函数体
```

```

        return e;
    }
}
var a = f(5());           //连续调用函数 f(), 而不是表达式的值
var b = f(5());           //连续调用函数 f(), 而不是表达式的值
alert( a === b )          //返回 false, 说明它们是不同的实例

```

看来形成上面奇特的闭包引用结果, 有其存在的必要条件:

- ☑ 函数在调用之前已经被调用, 这样在调用之前已经创建了调用对象。
- ☑ 函数内部已经定义了函数, 而不是被调用之后才定义。这样返回的是定义函数的引用地址, 而不是匿名函数体本身。

下面这个改进示例可以帮助读者理解上面的两个条件:

```

function f( x ){           //普通函数
    var e = function(){    //内部匿名函数
        return x;
    }
    return function(){      //返回嵌套函数体
        return e;
    }
}
var f = f( 5 );            //调用函数 f(), 定义内部函数, 生成实例
var a = f();               //调用表达式的返回实例
var b = f();               //调用表达式的返回实例
alert( a === b )           //返回 true, 说明它们引用相同函数, 即实例函数

```

闭包与实例实际上存在紧密的联系, 可以说它们是两个相等的概念。函数实例是闭包结构的体现, 而闭包也是实例的特殊结构。函数实例都拥有自己的闭包, 但是也存在一个函数实例拥有多个闭包的特殊情况。

16.7.9 闭包函数和调用对象

函数与对象也存在本质的不同, 函数是可执行的环境, 即我们常说的执行环境 (执行上下文环境)。而对象则是对象上下文环境, 具有静态性。如果说对象是数据存储系统, 那么函数就是执行系统。对象更像是一个仓库, 而函数则像产品加工间。所以, 函数必须使用小括号进行调用, 然后计算函数体内代码, 并返回一个值。而对象就不会被运算, 它相当于一个成员列表结构, 通过点语法或冒号来存储和访问内部的数据。例如:

```

function f(){              //函数结构
    var a = 1;              //函数局部变量
    var b = function(){    //函数内部的方法
        return a;
    }
}
o = {                      //对象结构
    a:1,                   //对象属性
    b:function(){          //对象属性
        return a;
    }
}

```

通过上面函数和对象结构的比较, 可以很直观地看出函数和对象的不同, 两者的详细比较如表 16.3 所示。

表 16.3 函数与对象的比较

项 目	函 数	对 象
结构	包含参数变量、局部变量、内部函数	属性、方法
性质	表达式运算	数据存储
调用	小括号运算符	点号运算符
状态	动态	静态
返回值	有	无
I/O	通过参数和返回值实现 I/O 功能	通过点号或中括号直接存取成员值
I/O 方式	固定方式的 I/O 接口	比较灵活的存取访问方式
系统	封闭	封闭
生存环境	执行环境（执行上下文）	对象环境（静态上下文）
相同	局部变量	属性
相等	内部函数	方法

闭包函数实际上就是内部函数，准确地讲就是内部闭包函数。而调用对象就是当前调用函数的对象结构（Script Object）。该对象存储着函数内部所有的数据，如参数变量列表、局部变量列表、内部函数列表以及其他代码（可执行逻辑）等。

当函数被调用并开始执行时，所有局部变量初始化为 `undefined`，但是由于函数在预编译时已经被解释器进行语法分析，所有局部变量、参数变量都已经被检索，所以此时访问局部变量不会发生未定义的语法错误。例如：

```
function f(){           //定义函数
    alert( a );         //读取并显示局部变量，显示提示为 undefined
    var a = 1;          //为局部变量赋值
}
f();                   //调用函数
```

实际上，当函数再次调用时，函数也会重新初始化，相当于刷新函数一次。函数被调用返回之后，其内部的变量值不会被刷新，变量值依然被保存在调用对象属性内，这种特性实际上就是闭包函数的基本特征。例如：

```
function f(){           //定义函数
    var a = 1;          //局部变量赋值
    function set(x){    //内部函数，动态改变局部变量 a
        a=x;
    }
    function get(){     //内部函数，跟踪局部变量 a 的值
        return a;
    }
    set(10);            //动态改变局部变量 a 的值
    alert(get());       //返回 10，观察局部变量 a 的值依然在，并发生变化
}
f();                   //调用函数
```

函数 `f` 及其内部的所有信息都会在返回后被销毁，当然对于与外界存在联系的成员除外。但是，在函数的生命期内，其内部的成员值依然能够保存，不会被重置。

当然，对于函数实例来说，其生命周期的长短取决于是否存在活动引用。如果没有被引用，则调用返回后，调用对象被销毁，内存被回收。例如：

```
function f(){
    var a = 1;
```

```

    return a;
}
var e = f;           //引用函数
f();                 //调用函数 f
alert(e());          //函数 f 依然存在，返回提示 1

```

因此一旦一个函数被引用之后，它的生命周期就会自动拉长，JavaScript 解释器能够自动检测并判断函数的外部关系。如果存在外部引用，则当该函数被引用之后，不会即时被注销，而是继续保存该函数的调用对象。而对于全局对象来说，由于它的特殊性和唯一性，决定了它也是一个例外，不能够与内部函数的调用对象相提并论。由于该全局函数无法重载，也不会被重新初始化，因此其内部的数据和结构本身不会被注销。

16.7.10 闭包独立性

闭包内的数据总是与外层数据保持动态联系，导致闭包数据在多个函数实例中保持独立性。先看一个示例，该示例设想在一个循环结构中通过闭包来自动更改一个数组内所有元素的值。代码如下：

```

var a = [1,2,3];           //定义并初始化数组
for(i in a){                //遍历数组
    a[i] = function(){      //通过闭包改变数组元素的值，使每个数组元素的值为该元素下标的平方值
        return i*i;
    }
}
alert(a[0]());              //返回 4
alert(a[1]());              //返回 4
alert(a[2]());              //返回 4

```

运行结果发现，数组的值都相同，为最后一个元素下标数的平方。原来在遍历操作中，闭包中变量 *i* 的值共享外部变量 *i* 的值，它们都是对于同一个值的不同引用。

如果检测 3 个函数实例，结果发现它们并不相等，则说明这 3 个函数实例是相互独立的：

```

alert(a[0]===a[1])          //返回 false，说明它们为不同的函数实例
alert(a[1]===a[2])          //返回 false，说明它们为不同的函数实例
alert(a[0]===a[2])          //返回 false，说明它们为不同的函数实例

```

也许，可以这样做：

```

for(i in a){
    a[i] = i*i;              //直接表达式赋值
}

```

这种做法是正确的，但是在特定的环境中，要求必须赋值为函数体时，如为事件属性赋值，此时就必须使用闭包函数了。代码如下：

```

for(i in o){
    o[i].onclick = function(){ //在事件属性赋值中，必须使用闭包函数
    }
}

```

解决这个问题的方法是为闭包函数再包裹一层函数体，因为函数结构具有存储数据的天性。把局部变量作为参数传递给函数，则函数就会把该参数作为私有数据进行保护，从而防止闭包内的数据与外层数据建立动态联系。代码如下：

```

var a = [1,2,3];
for(i in a){
    a[i] = function(i){       //包裹闭包体的带参数函数
        return function(){    //返回闭包体
            return i*i;
        }
    }
}

```



```

    }(i)                                //把局部变量传递给函数
  }
  alert(a[0]());                        //返回 0
  alert(a[1]());                        //返回 1
  alert(a[2]());                        //返回 4
  alert(a[0]===a[1])                    //返回 false, 说明它们为不同的函数实例
  alert(a[1]===a[2])                    //返回 false, 说明它们为不同的函数实例
  alert(a[0]===a[2])                    //返回 false, 说明它们为不同的函数实例

```

通过在闭包外面包裹一层函数来实现闭包数据的单独存储, 也存在一定的问题。因为多了一层闭包函数, 将增大系统的负担。不过, 通过上面示例的代码可以发现, 每次遍历中, 所定义的闭包结构都属于不同的函数实例。既然它们属于不同的结构体, 则可以把这个局部变量值交给函数实例自己保存, 从而减少了一层闭包结构, 减轻了系统的负担, 这在大循环中效果非常明显。代码如下:

```

var a = [1,2,3];
for(i in a){
    a[i] = function(){                //定义匿名函数
        var i = arguments.callee.value; //获取匿名函数的属性 value 值
        return i*i;                  //设置匿名函数的返回值
    }
    a[i].value = i;                    //设置匿名函数的属性 value 值为局部变量 i
}
alert(a[0]());                        //返回 0
alert(a[1]());                        //返回 1
alert(a[2]());                        //返回 4
alert(a[0]===a[1])                    //返回 false, 说明它们为不同的函数实例
alert(a[1]===a[2])                    //返回 false, 说明它们为不同的函数实例
alert(a[0]===a[2])                    //返回 false, 说明它们为不同的函数实例

```

在循环体内定义匿名函数, 在该函数体内获取函数属性 value 的值, 然后在闭包体的底部定义函数的属性 value 的值为局部变量的值, 通过这种方式把局部变量传递给函数实例的属性 value; 在匿名函数内调用函数属性即可实现保护闭包体内数据的独立性目标。

16.7.11 构造函数闭包

一般使用匿名函数或函数直接定义闭包函数, 实际上构造函数也能够定制闭包结构体。构造函数闭包不管在什么位置创建, 它都属于全局结构, 即作为 Function()构造函数的实例而存在。

例如, 在下面这个示例中, 在函数 f()中定义一个构造函数实例 e(), 然后在函数体调用该实例函数, 则返回值为 1, 而不是局部变量 2。

```

var a = 1;                            //全局变量 a, 初始化为 1
function f(){
    var a = 2;                          //局部变量 a, 初始化为 2
    var e = new Function("alert(a)");   //定义构造函数实例
    e();                                //调用构造函数实例
}
f();                                    //调用函数 f(), 返回值为 1, 而不是 2

```

由于构造函数的参数为字符串, 在编译期间不会被解析, 因此也不会与局部变量发生任何关系。但是由于构造函数是全局对象成员, 作为全局对象的闭包只会引用全局变量的值。

因此, 可以在复杂的函数嵌套结构中, 通过构造函数创建闭包, 从而使闭包体能够即时释放, 避免闭包之间数据相互干扰。例如, 下面的示例通过 Function()构造函数来闭包数据不能够相互保持独立的特性。每次解析 Function()构造函数时, 会重新为函数实例存储数据, 从而阻断了闭包数据之间的相互干扰性。代

码如下:

```
var a = [1,2,3];           //定义数组
for(i in a){               //遍历数组
    a[i] = new Function("return "+i*i); //通过构造函数闭包体来传递数据
}
alert(a[0]());             //返回 0
alert(a[1]());             //返回 1
alert(a[2]());             //返回 4
alert(a[0]===a[1]);        //返回 false, 说明它们为不同的函数实例
alert(a[1]===a[2]);        //返回 false, 说明它们为不同的函数实例
alert(a[0]===a[2]);        //返回 false, 说明它们为不同的函数实例
```

使用 Function() 构造函数创建函数, 系统就不会维护多个闭包, 也不会函数实例中绑定多余的成员, 这样就避免了闭包泛滥所带来的资源灾难, 避免内存外溢。关于这个问题我们会在后面的小节中进行详细讲解。当然, 由于 Function() 构造函数的参数必须是字符串, 这在一定程度上限制了它的应用范围, 因为在解析 Function() 构造函数的参数时, 首先要把所有变量都转换为字符串, 使用字符串作为参数有时会改变变量的原来意图, 特别是引用类型的变量。

当然, 使用 Function() 构造函数可以产生多个函数实例, 但是却不会产生多个闭包, 从而降低系统维护多个闭包所占用的资源。代码如下:

```
function f(x){             //普通函数
    var str = "return " + x + ";"; //定义函数结构字符串
    return new Function(str); //返回构造函数实例
}
var a = f(1);              //函数实例 1
var b = f(2);              //函数实例 2
alert(a());                //返回 1
alert(b());                //返回 2
alert(a===b);              //返回 false, 说明 a 和 b 属于不同的函数实例
```

上面的用法可以使用闭包函数来表示, 但是它将会产生多个闭包体, 从而占用大量系统资源。代码如下:

```
function f(x){             //外部函数
    return function(){     //内部函数
        return x;
    }
}
var a = f(1);              //函数实例 1
var b = f(2);              //函数实例 2
alert(a());                //返回 1
alert(b());                //返回 2
alert(a===b);              //返回 false, 说明 a 和 b 属于不同的函数实例
```

16.7.12 应用闭包函数

闭包常见的用法就是要为要执行的函数提供参数。例如, 为事件属性传递动作, 为定时器函数传递行为等。这在 Web 前端中是非常常见的一种应用。下面看一个示例:

```
function f(a,b){           //定义函数
    return function(){     //返回闭包函数
        a(b);
    }
}
var c = f(alert,"Hello,World"); //调用函数 f()
var d = setTimeout(c,1000);    //把闭包作为参数进行传递
```

预定义函数 `setTimeout()` 用于有计划地执行一个函数，或者一串 JavaScript 脚本，要执行的函数是其第 1 个参数，第 2 个参数是以毫秒表示的执行间隔。也就是说，当在一段代码中使用 `setTimeout()` 函数时，需要将一个函数的引用作为它的第 1 个参数，而将以毫秒表示的时间值作为第 2 个参数。但是，传递函数引用的同时就无法调用函数。类似下面的用法是错误的：

```
function f(a,b){
    a(b);
}
var c = f(alert,"Hello,World");
var d = setTimeout(c,1000);           //返回第 1 个参数为非法的错误
```

然而，可以在代码中调用另外一个函数，由它返回一个对内部函数的引用，再把这个对内部函数对象的引用传递给 `setTimeout()` 函数。执行这个内部函数时要使用的参数在调用外部函数时进行传递，这样，`setTimeout()` 函数在执行内部函数时，不用再传递参数，但该内部函数仍然能够访问在调用外部函数时传递的参数。

再看一个示例，该示例演示了如何使用闭包作为值来进行传递。当文档加载完毕后，会自动弹出一个提示对话框。其中正是利用闭包来实现向 Window 对象的 `onload` 属性传递一个闭包函数的，从而实现动态调用的效果。代码如下：

```
function f(a,b){
    return function(){
        a(b);
    }
}
var c = f(alert,"Hello,World");           //调用函数 f()
window.onload = c;                       //把闭包作为值进行传递
```

闭包还可以用于创建额外的作用域，通过该作用域可以设计动态数据管理器。利用动态数据管理器将相关的和具有依赖性的代码组织起来，以便应对复杂的交互操作。

例如，预设计一个字符串动态生成函数。该函数的功能是，把所有字符单元存储在一个数组中，通过数组方法把它们连接在一起并返回为一串字符串。如果说仅就一个数组进行操作，这种静态的、单一的处理就没有实际意义了。现在的问题是，我们希望数组中部分元素的值是动态的，然后把这个动态数组的元素值连接在一起生成一个字符串。

如果每次生成字符串时，都重新定义数组，那么也就没有必要去研究了，直接把数组作为参数传递给函数即可。现在仅更新数组的部分元素值，该如何做呢？

一种解决方案是将这个数组声明为全局变量，这样就可以重用这个数组，而不必每次都建立新数组。但这个方案的结果是，除了引用函数的全局变量会使用这个缓冲数组外，还会多出一个全局属性引用数组自身。如此一来，不仅使代码变得不容易管理，而且，如果要在其他地方使用这个数组时，开发者必须要再次定义函数和数组。这样一来，也使得代码不容易与其他代码整合，因为此时不仅要保证所使用的函数名在全局命名空间中是唯一的，还要保证函数所依赖的数组在全局命名空间中也必须是唯一的。

而通过闭包可以使作为缓冲器的数组与依赖它的函数关联起来，实施优雅的打包，同时也能够维持在全局命名空间外指定的缓冲数组的属性名，免除了名称冲突和意外交互的危险。代码如下：

```
var f = function(){           //函数表达式
    var a = [1,2,3,4,5,6,7,8,9,0] //数组初始值
    return function(a1,a2,a3,a4,a5){ //返回的闭包函数
        a[0] = a1;
        a[1] = a2;
        a[2] = a3;
        a[3] = a4;
        a[4] = a5;
```

```

    return a.join("-");           //返回的数组字符串
  };
})();                             //执行函数表达式，生成执行环境
var a = f(11,12,13,14,15);       //动态更新的值
var b = f("a","b","c","d","e");  //动态更新的值
alert(a);                        //返回 11-12 -13-14 -15- 6-7-8 -9-0
alert(b);                        //返回 a-b-c -d-e- 6-7-8 -9-0

```

其中关键的技巧就在于通过执行一个函数表达式创建一个额外的执行环境，而将该函数表达式返回的内部函数作为在外部代码中使用的函数。此时，缓冲数组被定义为函数表达式的一个局部变量。这个函数表达式只需执行一次，而数组也只需创建一次，就可以供依赖它的函数重复使用了。

上面的示例设计一个数组，该数组包含 10 个元素，其中最后 5 个元素的值都是静态的，每次创建动态字符串时，仅希望更新前面 5 个元素的值。这时就使用了闭包作为一个特殊作用域，然后该作用域与外部函数中的局部数组变量关联在一起。这样每次调用时，只需要动态向闭包函数传递动态更新的值，然后由闭包结构把更新的值传递给数组，并把数组生成为字符串返回即可。

下面再看一个闭包的典型应用。很多时候我们希望引用一个函数后能够暂停执行，因为在复杂的环境中不等到被执行的时候是很难知道其具体参数的，而先前被引用时更是无法预知所要传递的参数。

例如，希望为页面中特定的元素或标签绑定几个事情，使其能够在鼠标经过、离开和单击时呈现不同的背景颜色。代码如下：

```

element.onclick = function(){    //鼠标单击事件
    element.style.backgroundColor = "red"; //元素背景色为红色
}
element.onmouseover = function(){ //鼠标经过事件
    element.style.backgroundColor = "blue"; //元素背景色为蓝色
}
element.onmouseout = function(){  //鼠标移开事件
    element.style.backgroundColor = "transparent";
//元素背景色为透明
}

```

但是，现在还无法预知所要控制的元素。也许，可以定义一个函数，通过参数形式来定位预控制的标签，然后调用该函数即可。代码如下：

```

function f( name ){              //为指定标签绑定事件的函数
    var e = document.getElementsByTagName( name );
//获取指定标签引用指针
    if( e ){                      //如果指定的标签存在，则为其绑定各种事件处理函数
        for( var i in e ){        //遍历标签中每个元素
            e[i].onclick = function(){ //鼠标单击事件
                e[i].style.backgroundColor = "red"; //元素背景色为红色
            }
            e[i].onmouseover = function(){ //鼠标经过事件
                e[i].style.backgroundColor = "blue"; //元素背景色为蓝色
            }
            e[i].onmouseout = function(){ //鼠标移开事件
                e[i].style.backgroundColor = "transparent";
                //元素背景色为透明
            }
        }
    }
}

```

但是，这种做法比较原始，使用 JavaScript 函数来封装与特定 DOM 元素的交互。如果创建与不同 DOM

元素关联的任意数量的 JavaScript 对象，每个对象实例并不知道实例化它们的代码将会如何操纵它们。也就是说，把注册事件处理函数与定义相应的事件处理函数分离，不妨使用下面这种方法。

这个函数用于创建将自身与 DOM 元素关联的对象，DOM 元素的标签名作为构造函数的字符串参数。所创建的对象会在相应的元素触发 onclick、onmouseover 或 onmouseout 事件时，调用相应的方法。代码如下：

```
function f( name ){
    //关联指定标签的对象
    var e = document.getElementsByTagName( name );
    //获取指定标签的引用指针
    if( e ){
        //判断是否存在
        for( var i in e ){
            //遍历对象集合
            e[i].onclick = click //为对象绑定事件处理函数
            e[i].onmouseover = over //为对象绑定事件处理函数
            e[i].onmouseout = out //为对象绑定事件处理函数
        }
    }
}

f.click = function( event, element ){
    //事件处理函数
    element.style.backgroundColor = "red";
}

f.over = function( event, element ){
    //事件处理函数
    element.style.backgroundColor = "blue";
}

f.out = function( event, element ){
    //事件处理函数
    element.style.backgroundColor = "transparent";
}
```

也就是说，把每种事件处理的函数分离出来，单独定义，这样就能够实现代码的优化。这时会发现，由于事件属性只能接收函数结构，而无法直接传递参数。为此定义一个事件处理程序，能够把事件函数与实例对象关联在一起。在下面的代码中，使用一个返回闭包函数的方法，把外部指定的函数对象以及要绑定的方法进行封装和转换，从而实现复杂条件下轻松处理事件处理问题。

- ☑ 功能：把对象和方法捆绑为一个事件处理的函数。
- ☑ 参数：o 表示调用对象的实例（即触发函数），m 表示该对象的事件处理方法。
- ☑ 返回：闭包函数，该内部函数将把对象实例和方法封装为事件处理函数，并传递必要参数。

```
function f( o, m ){
    //实例对象与方法关联处理器
    return function( e ){
        //返回闭包函数，并传递事件句柄参数
        e = e || window.event;
        //事件对象兼容处理方法
        return o[m]( e, this );
        //返回一个函数调用，在该函数中把对象与事件方法进行绑定
    };
}
```

其中第 3 行代码表示，在支持标准 DOM 规范的浏览器中，事件对象会被解析为参数 e。如果是 IE 浏览器，则使用 IE 的事件对象来规范化事件对象。

第 4 行代码表示，事件处理器通过保存在字符串变量 m 中的方法名调用了对象 o 的一个方法，并传递已经规范化的事件对象和触发事件处理器的元素的引用 this，this 在这里指代该元素。

最后，完整的示例代码如下：

```
<script language="javascript" type="text/javascript">
function f( o, m ){
    //事件处理封装函数
    return function( e ){
        //返回闭包函数，将作为一个 DOM 元素的事件处理器
        e = e || window.event;
        //获取事件处理对象
        return o[m]( e, this );
        //返回闭包函数，利用传递的必要参数封装事件处理函数
    };
}
```

```

    }
}
function g( id ){                                //封装事件处理器函数，以实现在页面初始化事件中触发
    return function(){                          //返回事件处理器函数
        var e = document.getElementsByTagName( id );
        if( e ){                                //判断是否存在指定对象集合
            for( var i in e ){                  //变量对象集合
                e[i].onclick = f( g, "click" ); //调用关联处理器，把对象与方法捆绑到事件属性中
                e[i].onmouseover = f( g, "over" ); //调用关联处理器
                e[i].onmouseout = f( g, "out" ); //调用关联处理器
            }
        }
    }
}
g.click = function( event, element ){           //为事件处理封装函数定义额外的事件处理方法
    element.style.backgroundColor = "red";
}
g.over = function( event, element ){            //为事件处理封装函数定义额外的事件处理方法
    element.style.backgroundColor = "blue";
}
g.out = function( event, element ){             //为事件处理封装函数定义额外的事件处理方法
    element.style.backgroundColor = "transparent";
}
window.onload = g( "p" );                      //在页面初始化事件中绑定事件处理函数
</script>
<p>p1</p>
<p>p2</p>
<p>p3</p>

```

16.7.13 闭包副作用

闭包很容易创建，例如，出于习惯按照内部函数能完成多种任务的想法来使用内部函数。意外创建或乱用闭包可能导致严重的负面作用，而且也会影响到代码的性能，如内存泄漏问题。问题不在于闭包本身，如果能够真正做到谨慎地使用它们，会有助于创建高效的代码。使用内部函数会影响到效率。例如：

```

var a = "red";                                //全局变量
function f( o ){                              //外部函数
    if( o ){                                  //判断是否存在该对象
        o.onmouseover = function(){          //绑定事件处理函数
            this.style.backgroundColor = a;
        };
        o.onmouseout = function(){           //绑定事件处理函数
            this.style.backgroundColor = "transparent";
        };
    }
}

```

这是经常使用的内部函数作为 DOM 元素的事件处理器，为指定对象元素添加 onmouseover 和 onmouseout 事件处理器。

这样，无论什么时候调用函数 f()，都会创建一个新的内部函数，该函数实例通过赋值构成了一个闭包。如果只调用一两次函数 f()，并没有什么妨碍。但如果频繁使用该函数，就会导致创建许多截然不同的函数实例对象。由于每个内部函数都与外部的全局变量 a 保持联系，从而构成了无数个闭包，因此，也就影响了

系统的执行效率。当函数 `f()` 被调用的次数越多时，这种副作用的影响力就越大。大量生成的垃圾无法被即时回收，从而导致系统运行效率下降，甚至导致死机现象。

上面的代码并没有注意到内部函数由于与外部的变量存在联系，从而构成了闭包的事实。实际上，可以通过另一种方式来避免闭包的形成，即单独定义一个用于事件处理器的函数，然后将该函数的引用指定给元素的事件处理属性。这样，只需创建一个函数对象，而所有使用相同事件处理器的元素都可以共享对这个函数的引用。代码如下：

```
<script language="javascript" type="text/javascript">
var a = "red";                //全局变量
function f( o ){              //外部函数
    if( o ){
        o.onmouseover = over; //引用外部函数
        o.onmouseout = out;   //引用外部函数
    }
}
var over = function(){        //公共事件处理函数
    this.style.backgroundColor = a;
}
var out = function(){         //公共事件处理函数
    this.style.backgroundColor = "transparent";
}
window.onload = function(){    //绑定函数到页面初始化事件属性中
    var o = document.getElementById( "p1" );
    f( o )
}
</script>
<p id="p1">p1</p>
```

在上面的示例中，内部函数并没有作为闭包发挥应有的作用。在这种情况下，不使用闭包反而更加有效率，因为不用重复创建许多本质上相同的函数对象。不过，可以把事件处理函数定义为函数 `f()` 对象的私有方法，从而与外部的其他变量分离开来。代码如下：

```
<script language="javascript" type="text/javascript">
var a = "red";                //全局变量
function f( o ){              //外部函数
    if( o ){
        o.onmouseover = f.over; //引用函数的方法
        o.onmouseout = f.out;   //引用函数的方法
    }
}
f.over = function(){          //定义函数对象的方法
    this.style.backgroundColor = a;
}
f.out = function(){           //定义函数对象的方法
    this.style.backgroundColor = "transparent";
}
window.onload = function(){    //绑定函数到页面初始化事件属性中
    var o = document.getElementById( "p1" );
    f( o )
}
</script>
<p id="p1">p1</p>
```

另外，还应该注意，在 IE 早期版本中，由于 JavaScript 垃圾回收机制存在 Bug，即如果某些宿主对象

构成了循环引用，那么这些对象将不会被当作垃圾收集。此时所谓的宿主对象指的是任何 DOM 节点（包括 Document 对象及其后代元素）和 ActiveX 对象。如果在一个循环引用中包含了一或多个这样的对象，那么这些对象直到浏览器关闭都不会被释放，而它们所占用的内存同样在浏览器关闭之前都不会被回收利用。当两个或多个对象以首尾相连的方式相互引用时，就构成了循环引用。例如：

```
function o1(){           //对象 1
    this.a = 1;           //对象 1 属性 a
    this.b = function(){  //对象 1 方法 b 引用对象 2 中的属性 a
        return (new o2()).a;
    }
}
function o2(){           //对象 2
    this.a = 2;
    this.b = function(){
        return (new o3()).a;
    }
}
function o3(){           //对象 3
    this.a = 3;
    this.b = function(){
        return (new o1()).a;
    }
}
var a = new o1()         //实例对象 1
var b = new o2()         //实例对象 2
var c = new o3()         //实例对象 3
alert(a.b());            //返回 2
alert(b.b());            //返回 3
alert(c.b());            //返回 1
```

在上面的示例中，对象 1 的成员引用了对象 2，对象 2 的成员引用了对象 3，而对象 3 的成员又引用了对象 1。对于纯粹的 ECMAScript 对象而言，只要没有其他对象引用对象 1、2、3，也就是说，它们只是相互之间的引用，那么仍然会被垃圾收集系统识别并处理。但是，在 IE 早期版本中，如果循环引用中的任何对象是 DOM 节点或者 ActiveX 对象，垃圾收集系统则不会发现它们之间的循环关系与系统中的其他对象是隔离的并释放它们，最终它们将被保留在内存中，直到浏览器关闭。

闭包非常容易构成循环引用，它会消耗大量（甚至全部）系统内存。因此，应该避免形成循环引用。而在无法避免时，也可以使用补偿的方法，如使用 IE 的 `onunload` 事件清空（即赋值 `null`）事件处理函数的引用。

第17章

jQuery 框架透析之面向对象基础

( 视频讲解：4 小时 44 分钟)

在不同的编程语言中，复杂的数据结构有不同的称呼，如对象、字典、数组、队列、列表、哈希表等，这些结构有很多相似之处，但用法和功能通常各不相同。

对象（Object）是无序数据的集合，与数组一样都是复杂的数据容器。但两者结构不同，数据处理方式不同。如果说数组是线性结构，则对象应是离散结构，因为对象包含的数据没有下标位置。对象内的数据具有数组的部分特点，多个数据成员之间通过逗号进行分隔，每个数据成员都标识了一个名称，这种形式在其他语言中被称为数据字典。

数组为通过下标确定数据顺序的集合，而对象是已命名的数据集合，这些已命名的数据通常被称为对象的成员。对于 JavaScript 来说，成员主要包括属性和方法，属性是对象的私有变量，当然外界是可以访问的。这与函数内的私有变量不同，函数的私有变量对于外界来说是无法访问的，方法是对象的行为。

从另一个角度来分析，对象的数据结构实际上就是一个名/值对的集合，这种形式也可称为哈希表，不过这种列表结构却通过复杂的形式使 JavaScript 对象拥有强大的功能。

JavaScript 对象与函数有着紧密的联系，函数构造对象，函数也是对象，同时 JavaScript 借助函数和对象的紧密关系，创造了以原型继承为核心的面向对象编程风格。

17.1 定义对象

在 JavaScript 中，对象是一种基本数据类型（引用类型数据）。ECMA-262 对对象是这样描述的：对象是属性的无序集合，每个属性存放一个原始值、对象或者函数。从本质上来分析，对象应该属于一种无特定顺序的值类型的数组。下面以一段 JavaScript 代码为例进行比较：

```
var a = [           //定义数组
    x = 1,
    y = 2
]
var o = {           //定义对象
    x : 1,
    y : 2
}
alert(a[1]);        //返回 2，读取数组中下标为 1，即第 2 个元素的值
```

```

alert(o["y"]);           //返回 2，以中括号语法读取对象中名为 y 的属性值
alert(o.y);              //返回 2，以点语法读取对象中名为 y 的属性值

```

上面代码很直观地对比了数组和对象在结构和用法上的异同，通过比较会发现两者之间存在一定的联系和相似之处。

尽管 ECMAScript 如此定义对象，但是它更通用的定义是：对象是基于代码的名词表示，也就是说，在面向对象编程语言中一切都是对象（包括函数），对象是万物之源。这种设计思想与现实世界相联系，实际上面向对象的编程思想正是基于现实世界，模拟真实生活的一种方法和概括。

17.1.1 认识对象

OOP（Object Oriented Programming，面向对象的程序设计）是一种计算机编程思想，它的核心理念是程序都是由无数个对象组合而成的。这种设计思想容易实现 3 个目标：重用性、灵活性和扩展性。在 OOP 编程思想基础上，又延伸出了以下几个核心概念。

- ☑ 组件：编程单元，即对象或类。它是把数据和功能封装在一起，并能够运行的程序单元，相当于程序模块（或代码块），是实现程序结构化的基础。
- ☑ 抽象性：不关心内部实现和处理，只关心外部使用。组件能够忽略具体处理的信息，只关注信息处理的逻辑能力。
- ☑ 封装性：代码的封闭性。不允许外界访问组件内部信息，确保不会出现以不可预期的方式改变组件的内部状态，只有在提供接口的情况下才允许访问内部状态。同时组件也不会干扰外界事务，只有在指定方法中才会执行特定的任务。
- ☑ 多态性：多变性。在不同环境或条件下，引用组件所产生的结果存在不同。
- ☑ 继承性：能够复制。允许在现存的组件基础上创建子类组件。

面向对象的程序设计完全不同于传统的面向过程的程序设计，它大大降低了软件开发的难度，使编程变得更加简单。

对象（Object）由类（Class）定义，类是对象的模型（或模板）。

类为对象定义接口，接口为开发者访问对象的属性和方法提供了方便，还定义了对象内部的工作机制，工作机制就相当于工厂中的操作流程。

当使用类创建对象时，该对象被称为类的实例（Instance），这个过程可以形象描述为实例化过程。类能够创建多个实例，实例结构相同，行为也相似，并且可以独立处理数据，不需要实例之间或类进行支持。

JavaScript 不支持类的概念，但是间接支持类实现过程。通过定义构造函数，然后再实例化即可模拟出类的特征。

构造函数（Function）不是特殊的函数，它只不过是一个用于创建对象的普通函数。例如，下面的示例先定义了一个普通的空函数，然后使用 `new` 运算符实例化该对象，最后为实例对象定义属性和方法并进行调用。

```

function f(){                //定义空函数，构造函数
}
var o = new f();              //对象实例化
o.name = "张三";              //定义对象属性
o.saying = function(){        //定义对象方法
    return "Hi, World";
}
document.write(o.name + "说：" + o.saying() + " "); //访问对象

```

在浏览器中预览上面的示例，将会在窗口中显示“张三说：“Hi, World””文本信息。

在 JavaScript 中，对象成员被称为对象特性（Attribute），特性可以是原始值，也可以是引用值，但本质上它们都是数据。如果特性中存储的是函数类型的数据，则称为对象的方法（Method），否则该特性就是属

性 (Property)。

在一般语言中, 成员对象是对象, 它既是某一类成员, 又是另一个类类型的对象; 对象成员是指共同组成对象的要素, 包括数据成员 (对象的属性) 和成员函数 (对象的方法) 等。

17.1.2 定义对象

定义对象有多种方法, 常用方法如下所示。

☑ 通过构造函数创建对象。例如:

```
var o = new Object();    //创建普通对象
var d = new Date();      //创建时间对象
var r = new RegExp();    //创建正则表达式对象
var a = new Array();     //创建一个空的数组对象
var me = new MyClass();  //创建一个自定义对象
```

创建对象之后, 就可以使用点号运算符为其定义属性, 即为对象结构装入数据。例如:

```
var o = new Object();    //创建普通对象
o.a = "string";          //定义属性 a, 值为字符串 string
o.b = true;              //定义属性 b, 值为布尔值 true
```

使用 Object() 构造函数创建的对象是一个不包含任何属性和方法的空对象, 而使用内置构造函数创建的对象将会继承该构造函数的属性和方法。

例如, new Array() 创建了一个空数组对象, 但是这个新创建的对象具有数组操作的基本方法和属性, 如 length 属性可以获取该数组的元素个数, 而 push() 方法将会为该数组对象添加新元素。例如:

```
var o = new Array();      //创建一个空对象
alert(o.length);          //返回值 0, 说明当前数组为 0 个元素
var l = o.push(1,2,3);    //调用 push() 方法为该数组添加 3 个元素, 并返回新长度
alert(l);                 //返回值 3, 说明数组中包含 3 个新元素
```

☑ 通过对象直接量定义对象

通过大括号语法来实现, 大括号包含的是一个名/值对列表 (属性名与属性值), 名与值之间通过冒号隔开, 而成员之间通过逗号分隔。属性值可以是任意类型的数据, 但是属性名必须符合 JavaScript 标识符的语法规则。最后一个属性末尾不要逗号。使用对象直接量显得更加灵活而直观, 因此成为开发人员使用最广的一种方法。例如:

```
var o = {                  //对象直接量
  a: 1,                    //定义属性
  b: true                  //定义属性
}
```

对象的属性与 JavaScript 变量基本相似, 但是变量名是标识符, 而在对象直接量中属性名仅是一个字符串标签。所以, 对于上面示例中定义的对象直接量, 也可以这样来表示:

```
var o = {                  //对象直接量
  "a": 1,                  //定义属性
  "b": true                //定义属性
}
```

此时, 对象的属性名仅是一个字符串标签。但是在使用构造函数创建对象时, 就不能够使用字符串标签来命名对象的属性名, 因为此时属性已经变成了对象的成员, 成员是合法的标识符。对象的属性可以是任意类型数据, 如值类型数据、数组、对象、函数等。

如果属性值是函数, 则该属性就成为对象的方法, 读取这个特殊的属性值时, 就必须附加小括号运算符。例如:

```
var o = {                  //对象直接量
  a: function()            //属性值为函数
}
```

```

    return 1;
  }
}
alert(o.a());           //附加小括号读取属性值，即调用方法

```

如果属性值是对象，则可以设计连续使用点号运算符引用内层对象的属性值。例如：

```

var o = {                //对象直接量
  a: {                   //属性值为对象
    b: 1
  }
}
alert(o.a.b);           //连续使用点号运算符读取内层对象的属性值

```

如果属性值是数组，则必须使用数组下标来读取某个元素的值。例如：

```

var o = {                //对象直接量
  a: [1,2,3]             //属性值为数组
}
alert(o.a[0]);          //使用下标来读取属性包含的元素值

```

读取对象的属性值，可以调用点语法来实现，如上面示例所示。还可以使用关联数组来实现，即通过字符串下标来读取指定属性的值。例如：

```

var o = {                //对象直接量
  a: 1
}
alert(o["a"]);          //使用关联数组来读取对象的属性值

```

17.2 使用对象

对象是一类数据，但是它也是一个操作实体，利用对象的属性和方法可以完成复杂的任务。下面讲解 JavaScript 对象的一般使用方法。

17.2.1 引用对象

对象是不能够直接访问的，也就是说用户无法访问对象在内存中的物理空间，只能通过访问对象的引用来间接地访问对象。当创建对象时，存储在变量中的值只是对象的引用指针（如图 17.1 所示），而不是对象本身。

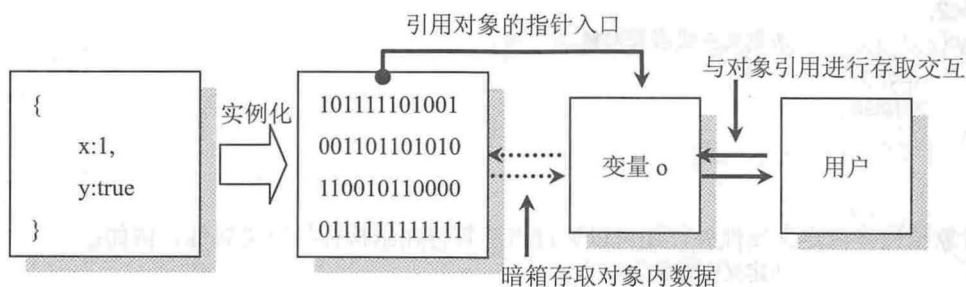


图 17.1 对象引用的示意图

例如，下面是对一个实例化对象的引用和删除操作。当删除对象引用的变量 `o` 之后，对象本身依然存在，并没有因为删除变量 `o` 而同时被删除。代码如下：


```

o = {                                //创建对象，并引用该对象给变量 o
    x:1,
    y:true
}
o1 = o;                             //复制变量 o
alert(delete o);                    //删除变量 o，返回值为 true，说明删除成功
alert(o1.x);                        //再次读取对象内数据，显示为 1，说明对象依然存在
alert(o.x);                        //利用 o 来读取对象内的数据，提示没有定义对象

```

17.2.2 销毁对象

JavaScript 语言有一套垃圾回收机制，这是一个能够自动收集内存中无用存储单元的程序。因此，不需要开发人员专门销毁无用对象以释放内存。当对象没有被任何变量引用时，JavaScript 引擎自动侦测并运行垃圾回收的程序把所有废除的对象注销，以释放内存。

每当函数对象被执行完毕，垃圾回收程序就会自动被运行，释放函数所占用的资源，并释放局部变量。另外，如果对象处于一种不可预知的情况下时，也会被回收处理。如果把对象的所有引用变量都设置为 null 时，可以强制对象处于废除状态，并被回收。例如：

```

var o = {                            //创建对象，并引用该对象给变量 o
    x:1,
    y:true
}
o = null;                           //定义对象引用变量为 null，即废除对象
alert(o.x);                         //提示系统错误，找不到对象

```

如果一个对象有多个引用，则必须将所有对象引用变量都设置为 null，只有这样对象清除才是成功的。在设计中，对于不用的对象，应该把其所有引用变量都设置为 null，将对象废除，用来释放内存空间。这是一种好的设计习惯，既节省了系统开支，同时也可以预防程序设计错误的发生。

17.2.3 定义对象属性

使用冒号逻辑标识符可以为对象定义属性，冒号左侧是属性名，右侧是属性值。属性与属性之间通过逗号运算符进行分隔。例如，下面示例定义了一个三层嵌套的对象结构体。虽然三层嵌套对象都包含相同的属性名，但由于它们分属于不同的作用域，因此不会发生名称冲突。代码如下：

```

var o = {                            //定义一级对象
    x:1,
    y:{                              //定义二级嵌套对象
        x:2,
        y:{                          //定义三级嵌套对象
            x:3,
            y:false
        }
    }
}

```

除了在对象结构体内定义属性外，还可以通过点运算符在结构体外定义属性。例如：

```

var o = {}                          //定义空对象
o.x = 1;                            //定义对象的属性
o.y = {                             //定义对象的属性，该属性的值是一个嵌套对象
    x: 2,
    y: true
}

```

另外，还可以通过构造函数来定义属性。例如：

```
var o = function(){ //定义构造对象
    this.x = 1;      //定义对象属性
    this.y = {       //定义包含对象的属性
        x:1,
        y:true
    }
}
```

在声明变量时使用 var 语句，但是在声明对象属性时是不能够这样做的。

17.2.4 访问对象属性

访问对象的属性通过点运算符来实现，左侧是对象引用的变量，右侧是属性名，属性名必须是一个标识符，而不是一个字符串。例如，针对 17.2.3 节示例中定义的三级嵌套对象，如果访问最里层对象的属性 y 的值，代码如下：

```
alert(o.y.y.y), //读取第 3 层对象的属性 y 的值，返回 false
```

对象属性与变量的工作方式相似，性质相同，可以把属性看作是对象的私有变量，用来存储数据。

从数据结构上来分析，对象与数组非常相似，因此可以使用中括号来访问对象的属性。例如，针对上面示例中读取最内层对象的属性 y，可以使用如下方式来读取：

```
alert(o["y"]["y"]["y"]), //读取第 3 层对象的属性 y 的值，返回 false
```

以数组形式读取对象属性值时，应该以字符串的形式指定属性名，而不能使用标识符。

可以使用 for/in 语句来遍历对象的属性。这对于无法确定对象长度时，是很有用的。例如：

```
var o = { //定义对象
    x:1,
    y:2,
    z:3
}
for(var i in o){ //遍历对象属性
    alert(o[i]); //读取对象的属性值
}
```

此时，应该使用数组操作方式来读取对象的属性值。使用 for/in 语句遍历对象属性时，是没有固定顺序的，同时它只能枚举用户自定义的属性，却无法枚举某些预定义的属性和方法。如果读取不存在的属性值，则返回值为 undefined，而不会引发系统错误。

17.2.5 操作对象属性

与读取对象属性值操作相同，设置对象属性的值也可以使用点运算符和数组操作方法来实现。例如：

```
var o = { //定义对象
    x:1,
    y:2
}
o.x = 2; //设置属性的新值，将覆盖原来的值
o["y"] = 1; //设置属性的新值，将覆盖原来的值
alert(o["x"]); //返回 2
alert(o.y); //返回 1
```

当为属性赋值，则对象就会自动创建属性，在任何时候和位置为该属性赋值，都不需要再创建这个属性，而只会重新设置它的值。

对于属性来说，可以使用 `delete` 运算符删除它，这与变量的操作相似。例如：

```
var o = { x:1}           //定义对象
delete o.x;              //删除对象的属性 x
alert(o.x);              //返回 undefined
```

当删除对象属性之后，不是将该属性值设置为 `undefined`，而是从内存中真正清除掉。如果使用 `for/in` 语句枚举对象的属性，则能够枚举属性值为 `undefined` 的属性，但是不会枚举已删除的属性。

17.2.6 操作对象方法

方法（Method）就是对象包含的另一类重要成员。从数据的角度来分析，JavaScript 对象的方法实际上就是一类特殊的属性，即属性值为函数的属性。从功能的角度来分析，方法是对象执行特定行为的逻辑块，是与外界实现行为交互的动作；而属性只是对象的变量，用来存储数据，或者与外界进行数据交流的接口。从语法的角度来分析，方法其实就是对象调用的函数。

在 JavaScript 中，由于函数也是一类数据，它可以被赋值给对象的任意属性。例如：

```
var o = {
  x:function(){          //定义对象的方法
    alert("method");
  }
}
```

或者，通过如下方式定义对象的方法：

```
var o = {}
o.x = function(){        //定义对象的方法
  alert("method");
}
```

定义方法之后，就可以采用读取属性的方式并结合小括号来执行方法：

```
o.x();                   //调用对象的方法
```

任何方法内部都包含一个 `this` 关键字，该关键字的值总是指向调用该方法的对象的引用指针。例如，在对象 `o` 的 `x()` 方法中访问当前对象的 `y` 属性值。当使用不同对象调用时，则返回值也不相同。代码如下：

```
var o = {
  x:function(){          //定义对象的方法
    alert(this.y);        //访问当前对象的属性 y 的值
  }
}
o.x();                   //返回 undefined，此时 this 指向对象 o
var f = o;               //复制对象 o 为 f
f.y = 2;                 //为对象 f 单独定义属性 y，赋值为 2
f.x();                   //返回 2，此时 this 指向对象 f
```

既然说对象的方法是函数，那么就可以在方法中传递参数，并返回值。实际上方法与函数在用法上没有什么不同。真正的差别就在于设计和目的上，方法是用 `this` 关键字指定的对象进行操作的，而函数往往都是独立的，没有 `this` 这个关键字来指定函数的操作对象。下面来看一个有关对象的方法的示例：

```
function f(){            //定义方法
  return this.x + this.y;
}
function MyClass(x,y){   //自定义类
  this.x = x;
  this.y = y;
}
var o = new MyClass(10,20); //实例化类，并初始化参数值
```

```
o.add = f;           //为实例对象绑定方法
alert(o.add());      //调用方法, 返回值 30
```

构造函数是一类没有返回值的函数, 它只是初始化对象, 利用参数来初始化 `this` 关键字所引用对象的 `x` 和 `y` 属性值。这样构造函数就相当于一个基本模板, 利用这个类可以实例化具有不同属性值的对象。可以把方法封装在类结构中, 这样就避免了为每个实例绑定方法的繁琐操作。例如:

```
function f(){        //定义方法
    return this.x + this.y;
}
function MyClass(x,y){ //自定义类
    this.x = x;
    this.y = y;
    this.add = f;      //把方法封装在类中, 这样每个示例都拥有了该方法
}
var o = new MyClass(10,20); //实例化类, 并初始化参数值
alert(o.add());           //调用方法, 返回值 30
```

17.3 对象作用域

在面向对象的程序设计中, 对象被视为一个封闭的作用域。访问对象的属性和方法是有严格限制的, 哪些属性或方法可以被公开访问, 哪些属性和方法仅在对象内部使用 (即私有成员), 都可以通过事先声明来确定, 这个访问权限的问题就是对象的作用域。

JavaScript 不支持对象作用域, 对象成员没有公开和私有之分, 任何成员都可以被外界访问, 不过在程序开发中, 可以通过一些非强制性的措施模拟这些对象作用域。

17.3.1 公共作用域

在很多强类型语言中, 对象成员的作用域会被分成很多等级。例如, 在 C# 语言中, 类的作用域被划分为以下 4 种。

- ☒ `public`: 公共的, 可以允许被外部成员调用。即可以在整个系统的任意地方调用, 是完全公开的。
- ☒ `private`: 私有的, 只能被类的成员调用。只能在类内部调用, 任何实例都无法调用私有成员。
- ☒ `internal`: 内部的, 可以在当前项目调用。即同一个项目, 这里的项目是单独的项目, 而不是整个解决方案。
- ☒ `protected`: 受保护的, 只能被类的成员和该类的子类调用。

但是 JavaScript 仅支持一种作用域, 即公共作用域。对于 JavaScript 所有对象的所有属性和方法来说都是公开的。例如:

```
var o = new Object() //创建对象, 并把对象引用存储在变量 o 中
o.x = new Object()   //创建对象, 并把对象引用存储在对象 o 的属性中
o.x.a = 1;           //声明对象 x 的属性 a, 并赋值
o.x.b = {             //创建对象, 并把对象引用存储在对象 x 的属性 b 中
    c: true,
    d: false
}
alert(o.x.b.c)        //访问内部对象的属性值, 返回 true
```

只要设置正确的访问路径, 在 JavaScript 中任何对象的任意属性和方法都可以访问。JavaScript 对象的结构封闭性不及函数结构。函数有作用域 (即局部作用域), 外界是无法访问的。所以, 从这个角度来辨析, 对象与函数最大的区别应该是它们的封闭性不同。

17.3.2 私有作用域

JavaScript 语言不支持对象的私有作用域。不过 JavaScript 开发人员积累了一套有效定义作用域的模式。开发者们可以非强制性约定：所有私有属性和方法，在命名时增加下划线前缀和后缀，以便与公共属性或方法进行区分。例如：

```
var o = new Object()
o.x = new Object()
o.x._a_ = 1;           //标识为私有属性
```

这种约定没有强制性，仅是为了有效区分公共成员和私有成员，不会改变 JavaScript 对象的所有成员性质。

17.3.3 静态作用域

当程序初始化之后，静态对象会自动被引入，且常驻内存以备对其进行引用，也就是说，内存中有一块区域是专门用来存放静态对象的属性、方法等成员变量的，需要时可以直接使用，不需要实例化操作。一般语法规则是：凡是声明为 `static` 的对象都是静态对象。

静态对象会长期占用系统资源，但是执行效率比较高。在 JavaScript 中，内置对象 `Math` 和 `Global` 都是静态对象，它们在 JavaScript 环境初始化时已经被引入内存。因此，在调用数学函数时，可以直接使用。同时，对于 JavaScript 程序中的各种变量、函数、自定义对象等也都可以直接使用。其原因就在于 `Global` 对象是一个静态对象，程序初始化时已经把这些变量引入内存，而不再需要进行实例化操作。

JavaScript 没有静态作用域。不过可以为构造函数定义属性和方法，来模拟这种静态作用域效果。例如：

```
function f(){           //定义普通函数
}
f.saying = function(){  //静态方法
    alert("static");
}
f.saying();             //直接调用方法
```

在上面的示例中，通过为函数定义方法，就可以直接进行调用，而不用实例化操作。实际上，方法 `saying()` 是函数 `f()` 的方法，而函数 `f()` 又是全局对象的一个方法，也可以直接调用。由于 `Global` 对象是静态对象，因此它包含的成员都属于静态作用域。当然，从本质上来分析，`saying()` 是 `f()` 公共作用域的方法，而不是静态作用域的方法，因此 JavaScript 不支持静态作用域这一特性。

17.3.4 对象指针 this

`this` 是面向对象中一个重要概念，它总是指向当前调用的对象。例如：

```
var o = new Object();    //对象实例化
o.name = "o";           //声明并初始化对象属性
o.who = function(){      //定义对象方法
    alert(this.name);     //显示当前对象的名称
}
o.who();                 //返回字符 o
```

在上面的示例中，`this` 指向对象 `o`，即定义该方法的对象自身。也可以直接引用对象本身。代码如下：

```
var o = new Object();    //对象实例化
o.name = "o";           //声明并初始化对象属性
o.who = function(){      //定义对象方法
    alert(o.name);       //显示对象 o 的名称
```

```

}
o.who();           //返回字符 o

```

在对象实例化的过程中，由于无法确定实例对象的名称，而使用 `this` 关键字能够确保在任意位置访问对象成员。例如：

```

function who(){           //定义一个抽象化方法
    alert(this.name);
}
var o = new Object();     //实例化对象 o
o.name = "o";             //命名为 o
o.who = who;              //引用抽象化方法 who
var f = new Object();     //实例化对象 f
f.name = "f";             //命名为 f
f.who = who;              //引用抽象化方法 who
o.who();                  //调用对象 o 的方法 who，返回字符 o
f.who();                  //调用对象 f 的方法 who，返回字符 f

```

在上面的示例中，首先使用 `this` 关键字定义一个公共方法，然后创建两个对象 `o` 和 `f`，分别设置它们的属性 `name` 值为 `o` 和 `f`，并绑定公共方法。但是分别调用不同对象的 `who` 方法时，返回值是不同的，这是因为关键字的作用。在对象 `o` 中，`this` 指向对象 `o` 自身。而在 `f` 对象中，`this` 又指向对象 `f` 自身，从而实现使用 `this` 关键字进行对象抽象化操作。

在公共方法中引用对象属性时，必须使用 `this` 关键字，否则 JavaScript 会把属性名看作是全局或局部变量，然后在该函数中查找该名为 `name` 的变量，显示这样是找不到的。

17.4 对象类型

根据功能不同，对象可以分为构造对象、原型对象和实例对象。构造对象和原型对象都是两类特殊的对象，但它们却是 JavaScript 编程的核心，实例对象是构造对象的实例，两者既相关联，也存在不同。

17.4.1 构造对象

构造对象实际上就是构造函数。构造函数与普通函数没有本质区别，不过它拥有 `this` 关键字，并且只能使用 `new` 运算符来调用函数。JavaScript 也预定义了一些构造对象，如 `Object`、`Date`、`Function` 等。

可以根据需要自定义构造对象，为了与普通对象相区别，构造对象的首字母一般建议大写。例如，下面定义了一个构造对象 `Point`，该对象包含两个属性 `x` 和 `y`。

```

function Point(x,y){      //构造对象
    this.x = x;           //对象属性
    this.y = y;           //对象属性
}

```

构造对象不能够直接被引用，必须使用 `new` 运算符来运算。例如：

```

var point = new Point(100,200); //实例构造对象

```

此时，对象 `point` 就可以被引用或存取操作了。

```

alert(point.x);           //读取对象的属性 x，返回值 100

```

1. 构造对象属性

构造对象就是类，类有属性和方法，因为类本身也是对象，它是一种抽象对象。构造对象的属性是一个与构造对象相关联的变量，而不是与由该类创建的每个实例对象相关联的变量。无论类创建多少个实例对象，每个类属性都只有一个副本。例如：

```
function Point(x,y){           //构造对象
    this.x = x;
    this.y = y;
}
Point.name = "类属性"         //构造对象的属性
var p1 = new Point(100,200);   //实例化对象
var p2 = new Point(300,400);   //实例化对象
alert(Point.name);             //直接读取构造对象的属性
alert(p1.name);                //但是不能够通过实例对象来读取构造对象的属性
```

在上面的实例中，演示了如何定义构造对象的属性，以及如何读取构造对象的属性值。与实例属性是通过实例对象来存取一样，构造对象的属性是通过构造对象来存取的。

对于构造对象的属性，实例对象是不能够继承的，因此也不能够通过实例对象来读取。反过来，也不能够通过构造对象来读取实例对象的属性。

构造对象的属性也被人们称为类的静态属性，即不通过实例化就可以直接读取的属性。

由于每个构造对象的属性都只有一个副本，所以构造对象的属性是全局变量。但是它们与类关联在一起，在 JavaScript 的名字空间中拥有一个逻辑位置，这样就不会被其他同名的全局变量所覆盖。

2. 构造对象方法

构造对象方法是一个与构造对象关联的方法，而不是与由该类创建的每个实例对象相关联的方法。要调用构造对象的方法，就必须使用构造对象本身，而不能使用该类的特定实例对象。例如，针对上面示例中的 Point 构造对象（或称为类），可以为它定义一个类方法：

```
Point.saying = function(){     //构造对象的方法
    alert("类方法");
}
Point.saying();                //直接调用构造对象的方法
```

构造对象的方法与构造对象的属性在本质上是相通的，即它们都是全局变量，也可以称之为静态方法。实例对象是不能够操作该方法的，所以构造对象的方法更容易被认为是由类调用的函数。同样地，将这些函数关联到一个类上可以使它们在 JavaScript 的名字空间中有一个独立的位置，避免与全局变量相冲突。

由于构造对象的方法不能通过一个实例对象调用，所以使用关键字 this 对它来说就没有意义，同时也不能够使用 this 关键字来读取实例对象的属性。例如，下面的写法是错误的：

```
Point.saying = function(){
    alert(this.x);
}
```

当然，并不是说在构造对象的方法中不能够使用 this 关键字，此时 this 指代的是 Point 对象，而不是对象实例。由于 Point 构造对象是 Function 构造对象的实例，所以下面的写法是合法的：

```
function Point(x,y){
    this.x = x;
    this.y = y;
}
Point.name = "类属性";
Point.saying = function(){
    alert(this.length);        //返回值 2
}
Point.saying();
```

在上面的示例中，this.length 表示读取函数 Point() 中包含的属性个数，所以返回值是 2，而不是 3。this 关键字在这里表示函数 Point() 的结构。属性 name 是构造对象 Point 的属性，而不是函数 Point() 的成员。

17.4.2 实例对象

实例对象是通过构造函数创建的，实例对象是构造对象的子对象，或者说是类（Class）的实例化（Instance）。创建实例对象可以通过 new 运算符来实例化构造函数即可。

1. 实例对象属性

实例对象的属性一般通过继承机制从构造函数自动获取，也可以单独定义实例属性。例如：

```
function Point(x,y){           //构造函数
    this.x = x;
    this.y = y;
}
var p =new Point(1,2);         //实例对象
p.z = p.x + p.y;               //自定义实例对象的属性 z
alert(p.z);
```

在上面示例中，实例对象 p 继承了构造函数 Point() 的两个属性 x 和 y，同时又自定义了一个属性 z。但是，继承属性具有共性，也就是说只要任何对象创建于 Point() 构造函数，都拥有该属性。当然它们的属性是不同的副本，而不是相同的引用。通俗地说，就是每个实例对象都有独立的实例属性副本。简而言之，如果有 10 个实例对象创建于同一个构造函数，那么每个实例属性就有 10 个副本。

而对于实例对象的自定义属性来说，其他实例对象是不会有，除非单独定义，或者引用该实例对象的自定义属性。在默认情况下，JavaScript 中任何对象的属性都是实例属性。但是为了真实模拟面向对象的程序设计语言，JavaScript 实例属性是那些在对象中用构造函数创建的或初始化的属性。

2. 实例对象方法

实例方法与实例属性在本质上都是相同的，它们都是对象的私有变量，只不过实例方法的值是函数，而不是原始值。实例方法使用了关键字 this 来引用它们要操作的对象或实例。例如，在下面的示例中，实例对象 p1 和实例对象 p2 的方法 saying() 分别引用各自对象的属性 x 的值：

```
function Point(x){
    this.x = x;
    this.saying = function(){
        alert(this.x);
    };
}
var p1 =new Point(1);
var p2 =new Point(2);
p1.saying();           //返回值 1
p2.saying();           //返回值 2
```

由于实例方法引用不同的副本，如果使用构造函数创建很多实例，那么就会产生很多个匿名函数副本，这样会降低系统资源的利用率。因此，在 JavaScript 中，给构造函数定义实例方法时，一般是通过把构造函数的原型对象中的一个属性设置为函数值来实现的。这样，由构造函数创建的所有对象都会共享同一个函数引用。

17.4.3 原型对象

当使用构造函数创建实例对象之后，每个实例对象都可以访问构造函数的原型对象，继承原型对象包含的属性和方法。原型对象为 JavaScript 提供了面向对象编程的继承方式。

1. 认识原型对象

原型对象实际上就是构造函数的一个实例对象，与普通的实例对象没有本质区别，不过它比较特殊，该对象包含的所有属性和方法能够供构造函数的所有实例共享，这就是其他语言中所说的类继承。而 JavaScript 通过原型对象来实现继承，简称为原型继承。

原型对象默认是构造函数自动产生的，不是人为定义的，但是可以读写，甚至可以覆盖原型对象。JavaScript 定义所有函数都有 `prototype` 属性，它指向原型对象，因此可以借助构造函数的 `prototype` 属性读写原型对象。

虽然原型对象初始化时是空的，但是为它定义的任何属性和方法都会被该构造函数创建的所有对象继承。例如，下面就是利用函数的 `prototype` 属性为构造函数 `Point` 定义了一个原型属性和原型方法：

```
function Point(x, y){           //构造函数
    this.x = x;
    this.y = y;
}
Point.prototype.name = "原型对象的属性";    //定义原型对象的属性
Point.prototype.sum = function(){           //定义原型对象的方法
    return this.x + this.y;                 //计算实例属性 x 和 y 的和，并返回
}
var p1 = new Point(1,2);                //实例对象
alert(p1.sum());                         //调用实例对象的原型方法 sum()，返回值为 3
alert(p1.name);                          //读取实例对象的原型属性
```

针对上面的示例，也可以这样定义原型属性和原型方法：

```
Point.prototype = {               //原型对象
    name : "原型对象的属性",      //原型对象的属性
    sum : function(){             //原型对象的方法
        return this.x + this.y;
    }
}
```

构造函数定义了对应的类，并初始化了类的属性，由于原型对象与构造函数紧密相连，所以类的每个实例对象都从原型对象继承了相同的属性。可以说，原型对象是存放类方法和其他常量的理想场所。使用原型对象的好处有以下两点：

- ☑ 使用原型对象可以大大降低对系统资源的消耗，因为所有实例对象都共享同一个原型对象，而不是各自拥有独立的副本。
- ☑ 在实例对象创建之后，仍然可以继续为对象添加原型属性和原型方法，这样所有实例对象都能够继承这些动态添加的原型属性和原型方法。例如：

```
var p1 = new Point(1, 2);
Point.prototype.name = "原型对象的属性";
Point.prototype.sum = function(){
    return this.x + this.y;
}
alert(p1.sum());                //调用实例对象的原型方法 sum()，返回值为 3
alert(p1.name);                 //读取实例对象的原型属性
```

上面的示例就是在对象实例化之后才定义原型属性和原型方法的，但是实例对象仍然能够继承这些后期动态添加的原型成员。当然下面的写法是错误的：

```
var p1 = new Point(1, 2);
Point.prototype = {
    name : "原型对象的属性",
    sum : function() {
```

```

        return this.x + this.y;
    }
}

```

因为这种写法实际上是覆盖了 `prototype` 属性对原型对象的引用。一个对象的原型是由创建并初始化该对象的构造函数定义的，也就是说，每当使用 `new` 运算符实例化构造函数时，JavaScript 会自动为对象传递一个原型对象的引用，原型对象的引用地址是在实例化过程中自动创建的，不管原型对象是否包含内容。一旦覆盖这个引用地址，即覆盖 `prototype` 属性值，JavaScript 解释器就无法准确找到原型对象包含的原型方法和原型属性。

2. 原型属性与实例属性

每个构造函数都有一个原型对象，原型对象中包含原型属性，每个实例对象都能继承原型属性。由于原型对象能够被多个实例对象继承，所以通过原型对象实现继承。但是，每个实例对象都可以自定义属性，原型属性与实例属性也会发生冲突，当发生冲突时，实例属性优先。

```

function Point(x, y){           //构造函数
    this.x = x;                 //对象属性
    this.y = function(){       //对象方法
        return y;
    }
}
Point.prototype.x = "a";       //原型属性
Point.prototype.y = function(){//原型方法
    return "b";
}
var p1 = new Point(1, 2);
p1.x = true;                   //自定义实例属性
p1.y = function(){            //自定义实例方法
    return false;
}
alert(p1.x);
alert(p1.y());

```

在上面的示例中，当 JavaScript 解释器在检索属性 `x` 和 `y` 时，首先会检查对象实例是否定义有名为 `x` 和 `y` 的属性。如果没有，则会再检查构造函数的原型对象是否具有这个属性。这样才使得以原型为基础的继承机制起作用，所以上面实例最后返回值为 `true` 和 `false`，其中自定义实例属性覆盖了初始化属性。但是，当写一个属性的值时，JavaScript 并不使用原型对象。因此，属性的继承只发生在读属性值时，而在写属性值时不会发生。

如果使用 `delete` 运算符删除实例对象的属性 `x` 和 `y`，此时弹出的信息是 `a` 和 `b`，这说明原型对象的属性是不能够被删除的。通过这种方法可以恢复对象的原型值，即利用原型对象为实例对象的属性设置默认值。当不需要默认值时，可以通过实例属性来覆盖原型属性即可。而如果需要默认值的对象，只需要删除自定义的实例属性即可。

```

var p1 = new Point(1, 2);
p1.x = true;
p1.y = function(){
    return false;
}
delete p1.x;                   //删除属性 x
delete p1.y;                   //删除属性 y
alert(p1.x);                   //返回原型属性值
alert(p1.y());                 //调用原型方法

```

3. 原型方法

JavaScript 所有构造对象都拥有原型对象，如 Function、Date、Object 等。利用原型方法为 JavaScript 内置对象扩展方法。例如，valueOf()方法是 Object 对象的一个默认方法，但是它的作用域是对象，而不是全局，因此必须按下面方法进行调用：

```
var a = "string";
alert(a.valueOf());
```

下面利用原型方法来扩展该方法的用法，使用该方法拥有全局作用域，也就是说，具有静态函数的功能。例如，可以按如下方式进行调用：

```
var a = "string";
alert(valueOf(a));
```

简单一看，好像没有什么差别，但是它们却属于不同对象的方法，也就是说，它们的作用域是不同的。具体实现的方法如下：

```
var valueOf = Object.prototype.valueOf = function(x){
    return x.valueOf();
}
```

上面的代码利用原型方法为 Object 构造对象定义了一个原型方法 valueOf()，这个原型方法虽然与 Object 对象的默认方法 valueOf()重名，但是它们的作用域是不同的，所以不会发生冲突。其中自定义的原型方法拥有全局作用域，而 Object 对象的默认方法 valueOf()仅拥有对象作用域。为了能够兼容 IE 浏览器，再定义一个全局变量引用这个原型方法，因为 IE 浏览器在全局作用域中找不到这个原型方法，而是把它作为对象作用域中的一个方法来对待。

17.4.4 构造器 constructor

构造器是面向对象编程的一个重要概念，它是用于创建实例的一个方法（构造函数）。JavaScript 没有类概念，但通过构造函数来表示类型，因此在 JavaScript 中构造器就指向了构造函数自身，而不是类型内部的一个方法。

为了方便引用构造函数，JavaScript 在 Object 超类中定义了一个 constructor 属性，即构造器属性，并定义该属性始终指向对象的构造函数。由于所有对象都继承于 Object 对象，所以每个对象都有一个 constructor 属性，它们分别引用所属的构造函数。例如：

```
function Point(){           //构造函数
var p1 = new Point();       //实例化对象
var o = p1.constructor;     //对象 p1 的构造器
alert(o == Point);         //返回 true，说明对象 p1 的构造器为 Point
```

构造函数 Point 也有构造器，因为构造函数也是对象，作为对象构造函数也会拥有自己的 constructor 属性。例如：

```
var p1 = new Point();
var o = p1.constructor.constructor;
alert(o);                 //返回 Function
```

通过上面示例可以看到构造函数 Point 的构造器为 Function 对象，也就是说，在 JavaScript 中 Function 对象是构造万物的基类。例如，下面的示例显示了超类 Object 对象，也是通过 Function 对象构造的。

```
var o = Object.constructor;
alert(o);                 //返回 Function
```

不过 Function 对象的构造器是它自身。

```
var o = Function.constructor;
alert(o == Function);     //返回 true
```

由于 Object 对象是所有对象的超类，作为对象的 Function，当然它也是 Object 的一个子类。


```
alert(Function instanceof Object); //返回 true, 说明 Function 对象是 Object 对象的实例
```

但是, 反过来 Object 对象也是 Function 对象的实例, 因为 Object 对象是由 Function() 函数构造的。

```
alert(Object instanceof Function); //返回 true, 说明 Object 对象也是 Function 对象的实例
```

由于 JavaScript 语言吸取各类语言的精华, 如函数式语言、面向对象语言等, 使其用法非常灵活。

原型对象也有构造器, 它的构造器指向原型对象的构造函数。例如:

```
var o = Point.prototype.constructor;
```

```
alert(o == Point); //返回 true, 说明原型对象的构造器为 Point
```

对象的构造器是通过原型对象进行传递的, 也就是说, 对象的 `constructor` 属性继承于原型对象。如果删除构造函数的原型对象, 那么它的所有实例对象就找不到继承的方法。此时, `constructor` 属性就沿着原型链, 指向更高级的构造器——Object 对象。例如:

```
function Point(){} //构造函数
```

```
Point.prototype = null; //删除构造函数 Point() 的原型对象
```

```
var p = new Point(); //实例化对象
```

```
var o = p.constructor; //获取实例的构造器
```

```
alert(o == Point); //返回 false, 说明对象 p 的构造器不是 Point
```

```
alert(o == Object); //返回 true, 说明对象 p 的构造器不是 Object
```

通过上面的示例, 也可以证明对象的构造器不是唯一的, 也就是说, `constructor` 属性值不是固定的。如果继续删除 Object() 构造函数的原型对象, 而对象 p 的构造器仍然指向 Object, 这是因为 Object 是所有对象的超类, 所有对象都继承于它。所以, 不管 Object 构造函数的原型对象是否存在, `constructor` 属性最后总是指向它。例如, 下面的示例演示了对象 p 的构造器最终指向 Object:

```
function Point(){} //构造函数
```

```
Point.prototype = null; //删除构造函数 Point() 的原型对象
```

```
Object.prototype = null; //删除构造函数 Object() 的原型对象
```

```
var p = new Point(); //实例化对象
```

```
var o = p.constructor; //获取实例的构造器
```

```
alert(o); //返回构造函数 Object()
```

由于构造器总是指向一个构造函数, 而构造函数定义了一个对象的类, 所以开发人员常使用属性 `constructor` 来判定对象的数据类型, 这是一个实用且准确的方法。例如, 下面条件语句可以确定一个变量是否为数字对象:

```
var n = new Number();
```

```
if(typeof n == "object" && n.constructor == Number){
```

```
    alert("该对象是一个数字对象");
```

```
}
```

但是, 对于自定义类来说, 并不能保证 `constructor` 属性总是存在的。例如, 在下面的示例中, 这个判定标准并不标准, 因为 `constructor` 属性被修改了, 它指向 Number() 构造函数的引用:

```
function Me(){} //自定义类
```

```
Me.prototype = new Number(); //修改该类的原型对象为数字对象
```

```
var m = new Me(); //实例化对象
```

```
if(typeof m == "object" && m.constructor == Me){
```

```
    alert("该对象是一个自定义类 Me 的对象");
```

```
}
```

不过, 对于 JavaScript 内置对象来说, 由于它们的原型是只读的, 用户无法修改它们的原型对象, 所以可以放心使用 `constructor` 属性判断内置对象的数据类型。

17.5 核心方法

在 JavaScript 中, Object 是超级数据类型, 所有对象都继承于 Object 对象, 因此也继承了 Object 对象的

属性和方法。在默认情况下, JavaScript 为 Object 对象预定义了几个属性和方法, 用户也可以通过原型对象为 Object 对象扩展属性和方法, 这样所有 JavaScript 对象都拥有了自定义的属性和方法。Object 对象默认定义了几个核心方法, 如表 17.1 所示。

表 17.1 Object 对象定义的基本方法

方 法	说 明
toString()	返回对象的字符串表示
toLocaleString()	返回对象的本地字符串表示
valueOf()	返回对象的原始值
isPrototypeOf()	判定一个对象是否为另一个对象的原型
hasOwnProperty()	检查属性是否被继承
propertyIsEnumerable()	判定是否可以通过 for/in 循环遍历对象的属性

17.5.1 toString()方法

toString()方法能够返回一个对象的字符串表示, 但是返回的字符串比较灵活, 它可能是一个具体的值, 也可能是一个对象的类型表示。例如:

```
function F(x,y){           //构造函数
    this.x = x;
    this.y = y;
}
var f = new F(1,2);        //实例化对象
alert(F.toString());       //返回函数的源代码
alert(f.toString());       //返回字符串[object Object]
```

由于 toString()方法比较简陋, 所以很多开发人员在自己的框架中对该方法进行扩展, 以便它能够返回更多有用信息。例如, 针对实例对象返回的字符串都是[object Object], 可以对其进行扩展, 让对象实例都能够返回构造函数的源代码。代码如下:

```
Object.prototype.toString = function(){
    return this.constructor.toString();
}
```

下面示例中的 f.toString()就会返回函数的源代码, 而不是字符串[object Object]。不过, 这种方法不会影响 JavaScript 内置对象的 toString()返回值, 因为它们都是只读的。

```
alert(f.toString());       //返回函数的源代码
```

当把数据转换为字符串时, JavaScript 一般都会调用 toString()方法来实现。由于不同类型的对象在调用该方法时, 所转换的字符串都不同, 而且都有规律, 所以开发人员常用它来判断对象的类型, 弥补 typeof 运算符和 constructor 属性确定对象类型的不足。

由于 toString()方法能够返回不同类型对象的特定的字符串, 所以当自定义数据类型时, 也应该定义一个合适的 toString()方法将对象转换成相应的字符串形式, 以方便准确跟踪自定义对象的数据类型。例如, 对于自定义类型的 toString()方法返回值为[object Object], 可以为自定义类型 Me 定义一个标志字符串[object Me]。代码如下:

```
function Me(){             //自定义数据类型
Me.prototype.toString = function(){ //自定义 Me 数据类型的 toString()方法
    return "[object Me]";
}
var me = new Me();
```

```

alert(me.toString());           //返回[object Me]
alert(Object.prototype.toString.apply(me)); //默认返回[object Object]

```

除了 `toString()` 方法外, JavaScript 还提供了 `toLocaleString()` 方法, 该方法能够返回对象的本地字符串。不过 `Object` 对象定义的默认 `toLocaleString()` 方法返回的结果与 `toString()` 方法返回的完全相同, 但是在不同子类型对象中可能会定义自己的 `toLocaleString()` 方法, 如 `Array`、`Date` 和 `Number` 对象类型。

17.5.2 valueOf()方法

`valueOf()` 方法返回对象的值。一般对象都没有定义为原始类型的值, 所以大多数对象都没有等价的原始值, 因此调用对象的 `valueOf()` 方法时, 返回值与 `toString()` 方法基本相同。同时, `Object` 对象的默认 `valueOf()` 方法返回值与 `toString()` 方法返回值是相同的, 但是一些内置对象重定义了该方法。例如, `Date` 对象的 `valueOf()` 方法返回值就是毫秒数。

```

Var o = new Date();           //对象实例
alert(o.toString());          //返回当前时间的 UTC 字符串
alert(o.valueOf());           //返回当前距离 1970 年 1 月 1 日午夜 (GMT 时间) 之间的毫秒数
alert(Object.prototype.valueOf.apply(o)); //Object 对象默认返回当前时间的 UTC 字符串

```

`String`、`Number` 和 `Boolean` 对象具有明显的原始值时, 它们的 `valueOf()` 方法会返回合适的原始值。因此, 在自定义数据类型时, 除了自定义 `toString()` 方法外, 不妨也预定义 `valueOf()` 方法。这样当读取自定义对象的值时, 避免返回的值总是 `[object Object]`。例如:

```

function Point(x,y){           //自定义数据类型
    this.x = x;
    this.y = y;
}
Point.prototype.valueOf = function(){ //自定义 Point 数据类型的 valueOf()方法
    return "(" + this.x + "," + this.y + " ";
}
var p = new Point(26,68);
alert(p.valueOf());             //返回当前对象的值(26,68)
alert(Object.prototype.valueOf.apply(p)); //默认返回值为[object Object]

```

在特定环境下转换数据类型时, `valueOf()` 方法的优先级要比 `toString()` 方法的优先级高。因此, 如果一个对象的 `valueOf()` 和 `toString()` 方法返回值不同时, 而又希望转换的字符串为 `toString()` 方法的返回值时, 就必须明确调用对象的 `toString()` 方法。例如, 在下面这个示例中, 当获取自定义类型的对象 `p` 时, `alert()` 方法会首先调用 `valueOf()` 方法, 而不是 `toString()` 方法。如果需要了解该对象的类型标志符, 则必须明确调用对象的 `toString()` 方法。

```

Function Point(x,y){           //自定义数据类型
    this.x = x;
    this.y = y;
}
Point.prototype.valueOf = function(){ //自定义 Point 数据类型的 valueOf()方法
    return "(" + this.x + "," + this.y + " ";
}
Point.prototype.toString = function(){ //自定义 Point 数据类型的 toString()方法
    return "[object Point]";
}
var p = new Point(26,68);           //实例化对象
alert("typeof p = " + p);           //默认调用 valueOf()方法进行类型转换
alert("typeof p = " + p.toString()); //直接调用 toString()方法进行类型转换

```

17.5.3 hasOwnProperty()方法

对象的属性分为两大类：私有属性和继承属性。例如，在下面的自定义类型中，`this.name` 就表示对象的私有属性，而原型对象中的 `name` 属性就是继承属性。

```
function F(){                                //自定义数据类型
    this.name = "私有属性";
}
F.prototype.name = "继承属性";
```

`hasOwnProperty()`方法可以快速、准确地确定指定属性的性质。例如，针对上面自定义数据类型，实例化对象，然后判定当前对象调用的属性 `name` 是什么性质。代码如下：

```
var f = new F();                            //实例化对象
alert(f.hasOwnProperty("name"));            //返回 true，说明当前调用的 name 是私有属性
alert(f.name);                             //返回字符串"私有属性"
```

凡是构造函数的原型属性，都是继承属性，使用 `hasOwnProperty()`方法检测时，都恢复返回 `false`。但是，对于原型对象本身来说，这些原型属性又是它们的私有属性，所以返回值又是 `true`。例如，在下面的示例中，演示了 `toString()`方法对于 `Date` 对象来说，是继承属性，但是对于 `Date` 构造函数的原型对象来说，则又是它的私有属性。代码如下：

```
var d = Date;
alert(d.hasOwnProperty("toString"));        //返回 false，说明 toString()方法不是 Date 对象的私有属性
var d = Date.prototype;
alert(d.hasOwnProperty("toString"));        //返回 true，说明 toString()方法是 Date. Prototype 对象的私有属性
```

`hasOwnProperty()`方法只能判断特定对象中是否包含指定名称的属性或对象，但无法检查对象原型链中是否具有某个属性，所能够检测出来的属性必须是对象本身的一个成员。例如，下面这个示例就比较直接地演示了 `hasOwnProperty()`方法所能够检测的属性范围。

```
var o = {                                    //对象直接量
    o1 : {                                  //子对象直接量
        o2 : {                             //孙子对象直接量
            name : 1                       //孙子对象直接量的属性
        }
    }
};
alert(o.hasOwnProperty("o1"));              //返回 true，说明 o1 是 o 的私有属性
alert(o.hasOwnProperty("o2"));              //返回 false，说明 o2 不是 o 的私有属性
alert(o.o1.hasOwnProperty("o2"));          //返回 true，说明 o2 是 o1 的私有属性
alert(o.o1.hasOwnProperty("name"));        //返回 false，说明 name 不是 o1 的私有属性
alert(o.o1.o2.hasOwnProperty("name"));     //返回 true，说明 name 不是 o2 的私有属性
```

17.5.4 propertyIsEnumerable()方法

枚举是一种数据类型，它表示被命名的整型常数值集合，这个集合可以允许使用循环结构遍历。在强类型语言中，由于一般集合（如对象、列表等）是不允许遍历操作的，只有通过转换为对应的枚举类型，才能够实现数据的快速检索。例如，下面的数据结构就是一个枚举类型：

```
enum Week {
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
```

```

Thursday = 4,
Friday = 5,
Saturday = 6
}

```

JavaScript 是一种弱类型语言，目前暂时还不支持枚举类型的数据结构，不过 ECMA-262 v3 保留了关键字 `enum`。但是 JavaScript 的对象实际上就是一个枚举类型的数据结构。例如，针对上面的枚举数据结构，在 JavaScript 中可以这样来设计：

```

if(typeof Week == "undefined"){           //检测 Week 变量是否存在
    var Week = {                          //定义对象直接量，结构类似枚举数据结构
        Sunday : 0,
        Monday : 1,
        Tuesday : 2,
        Wednesday : 3,
        Thursday : 4,
        Friday : 5,
        Saturday : 6
    }
}

```

对于 JavaScript 对象来说，可以枚举它的属性。但并不是对象的所有属性都是可以枚举的，只有满足了下面两个条件才行：

- ☑ 属性名是由一个字符串实际参数指定的，即对象的所有私有属性和原型属性。但是 JavaScript 又规定，只有对象直接定义的属性才可以枚举。使用 `for/in` 循环可以枚举原型属性。
- ☑ 属性可以使用 `for/in` 循环进行遍历读写。例如，JavaScript 核心对象的默认属性一般都不允许枚举的。

```

var o = new Object();
for(var i in o){
    document.write(i + "<br />");
}

```

对于下面这个自定义对象 `o`，虽然使用 `for/in` 循环可以遍历它的所有私有属性、原型属性，但是允许枚举的属性只有 `a` 和 `b`。

```

function F(){
    this.a = 1;
    this.b = 2;
}
F.prototype.c = 3;
F.d = 4;
var o = new F();
for(var i in o){
    document.write(i + "<br />");
}

```

使用 `propertyIsEnumerable()` 方法可以准确判定某个属性是否可以枚举。如果该方法的返回值为 `true`，说明指定的属性可以枚举，否则是不允许枚举的。例如：

```

alert(o.propertyIsEnumerable("a")); //返回值为 true，说明可以枚举
alert(o.propertyIsEnumerable("b")); //返回值为 true，说明可以枚举
alert(o.propertyIsEnumerable("c")); //返回值为 false，说明不可以枚举
alert(o.propertyIsEnumerable("d")); //返回值为 false，说明不可以枚举
var o = F;
alert(o.propertyIsEnumerable("d")); //返回值为 true，说明可以枚举

```


17.5.5 isPrototypeOf()方法

在 JavaScript 语言中，每个函数都有一个 `prototype` 属性，该属性指向一个原型对象。当声明或定义一个函数时，JavaScript 解释器会自动调用 `Object()` 构造函数为该函数创建一个空的实例对象，并传递给函数的 `prototype` 属性，当然也可以操作这个原型对象。原型对象仅供当前函数使用。例如：

```
var f = function(){} //定义函数
f.prototype = {      //函数的原型对象
  a : 1,
  b : function(){
    return 2;
  }
}
alert(f.prototype.a); //读取函数的原型对象的属性 a，返回 1
alert(f.prototype.b()); //读取函数的原型对象的属性 b，返回 2
```

但是当使用 `new` 运算符调用函数时，就会创建一个实例对象，这个实例对象将动态继承构造函数的原型对象的所有属性，并可以直接存取。从某种意义上讲，原型对象就是所有实例对象的公共存储空间。例如：

```
var o = new f(); //实例对象
alert(o.a);      //访问原型对象的属性
alert(o.b());    //访问原型对象的属性
```

`isPrototypeOf()` 方法能够准确判定一个对象的原型对象。例如，针对下面的示例，可以看到 `f.prototype` 就是对象 `o` 的原型对象，所以下面的示例就会返回 `true`：

```
var b = f.prototype.isPrototypeOf(o);
alert(b);
```

同理，下面的示例演示了各种特殊对象的原型对象。

☒ 函数的原型对象可以是 `Object.prototype` 或者 `Function.prototype`：

```
var f = function(){}
alert(Object.prototype.isPrototypeOf(f)); //返回 true
alert(Function.prototype.isPrototypeOf(f)); //返回 true
```

☒ `Object` 和 `Function` 对象的原型对象比较特殊：

```
alert(Function.prototype.isPrototypeOf(Object)); //返回 true
alert(Object.prototype.isPrototypeOf(Function)); //返回 true
```

☒ `Object.prototype` 和 `Function.prototype` 的原型对象都是 `Object.prototype`，而 `Function.prototype` 的原型对象还可以是 `Function.prototype`，但是 `Object.prototype` 的原型对象绝对不是 `Function.prototype`：

```
alert(Object.prototype.isPrototypeOf(Object.prototype)); //返回 true
alert(Object.prototype.isPrototypeOf(Function.prototype)); //返回 true
alert(Function.prototype.isPrototypeOf(Function.prototype)); //返回 true
alert(Function.prototype.isPrototypeOf(Object.prototype)); //返回 false
```

17.6 核心对象

在 JavaScript 中，`Window`、`Document` 是对象，`Object`、`Global` 也是对象，用户还可以自定义对象等。从语法角度分析上述对象，它们都属于一类数据结构。但是，如果细分这些 JavaScript 对象，则它们的级别是不同的。

17.6.1 对象系统

JavaScript 对象系统包括 3 种类型：本地对象、内置对象和宿主对象。

1. 本地对象

本地对象就是独立于宿主环境的 JavaScript 预定义对象，这些对象实际上都是构造函数，如表 17.2 所示。由于用户自定义的对象都是本地对象的具体实例，所以它们也应该属于本地对象范畴。

表 17.2 本地对象

Object	Function	Array	String
Boolean	Number	Date	RegExp
Error	EvalError	RangeError	ReferenceError
SyntaxError	TypeError	URIError	

2. 内置对象

内置对象是由本地对象实现的，且独立于宿主环境的所有对象。内置对象在 JavaScript 程序执行时，会自动初始化并存在。也就是说，能够常驻内存，因此不需要实例化内置对象。ECMA-262 标准只定义了 Global 和 Math 两个内置对象。从数据结构上来分析，内置对象与本地对象相同，不过内置对象是本地对象的一类特例。

本地对象和内置对象统称为原生对象（Native Object），与宿主对象（Host Object）相对而论。其中内置对象是原生对象的子类。原生对象属于 JavaScript 语言，而宿主对象却由具体的环境定义。

原生对象具有松散和动态的命名属性。对象的命名属性用于保存值，该值可以是指向另一个对象的引用。从这个意义上说，函数也是对象，也可以是一些基本的数据类型，如 String、Number、Boolean、null 或 undefined。其中比较特殊的是 undefined 数据类型，因为可以给对象的属性指定一个 undefined 类型值，则表示该属性未定义，系统不会删除。但是如果赋值为 null，则表示该属性不存在，系统会自动回收这些属性。

3. 宿主对象

如果不是 JavaScript 语言定义的对象都是宿主对象，宿主对象就是 JavaScript 寄宿环境定义的对象。例如，Window、Document、History 等都是宿主对象，它们由客户端浏览器环境定义，与 JavaScript 语言本身没有直接关系。不过 JavaScript 能够控制这些对象的行为，实现对其进行读写操作。

在 JavaScript 语言的核心模块中，定义了很多对象，这些对象是 JavaScript 开发的基础。如 Function 对象是原型继承的基础，而 Object 对象是对象结构的基础，而 Number、Boolean 和 String 对象又是基本数据类型的结构化封装工具。另外，Array、Error、Date、RegExp、Math 对象又是程序开发中不可或缺的。

17.6.2 Global 对象

Global 是一个虚拟对象，代表全局对象，在规范中 Global 只是一个概念，它是不存在，也是无法访问的，只有在具体宿主环境中才能够确定 Global 对象指代的具体对象，如在浏览器中全局对象是 Window。

```
alert(typeof Global);           //访问 Global 对象的类型，返回 undefined
var o = new Global();           //返回未定义对象的编译错误
```

Global 对象拥有众多属性（如表 17.3 所示）和方法（如表 17.4 所示），甚至 JavaScript 预定义的所有对象，包括用户自定义的所有对象、属性和函数等。

表 17.3 Global 对象属性

属 性	说 明
Infinity	表示正无穷大的数值
NaN	非数值
undefined	未定义的值

表 17.4 Global 对象方法

方 法	说 明
encodeURIComponent()	通过转义某些字符对 Url 进行编码
decodeURI()	对使用 encodeURIComponent()方法编码的字符串进行解码
encodeURIComponent()	通过转义某些字符对 URI 的组件进行编码
decodeURIComponent()	对使用 encodeURIComponent()方法编码的字符串进行解码
escape()	使用转义序列替换某些字符来对字符串进行编码
unescape()	对使用 escape()编码的字符串进行解码
eval()	计算 JavaScript 代码字符串, 并返回计算的值
isFinite()	检测一个值是否是无穷大的数字
isNaN()	检测一个值是否是数字的值
parseFloat()	把字符串数据解析为浮点类型的数据
parseInt()	把字符串数据解析为整型子类型的数据

Global 是预定义的全局对象, 作为 JavaScript 的全局属性和全局函数的概念而存在。通过使用全局对象, 可以访问其他所有预定义的对象、函数和属性。不过, 全局对象是对象, 而不是类。既没有 Global()构造函数, 也无法实例化一个新的全局对象。它包含所有属性、函数和对象, 但是全局函数却不是 Global 对象的方法, 全局属性也不是 Global 对象的私有属性。

```
var s = "alert('Hello,World!');"; //定义字符串
Global.eval(s);                //调用 Global 对象的方法 eval()执行字符串错误
eval(s);                       //直接调用全局函数 eval()有效
```

上面示例说明, Global 对象仅是一个抽象概念, 如果在全局作用域中, 使用关键字 this 能引用全局对象。例如:

```
var s = "alert('Hello,World!');"; //定义字符串
this.eval(s);                    //使用 this 对象调用 Global 对象的函数
```

一般很少使用这种方式来引用全局对象, 因为全局对象是作用域链的头, 这意味着所有全局变量、函数等都作为该对象的属性直接被访问。当 JavaScript 代码嵌入一个特殊环境中时, 全局对象又具有环境定义的属性。实际上, ECMAScript 也没有规范全局对象的类型, JavaScript 可以把任意类型的对象作为全局对象, 只要该对象定义了特殊环境列出的基本属性和函数。例如, 在客户端 JavaScript 环境中, 全局对象是 Window 对象, 它表示运行 JavaScript 代码的浏览器窗口。

17.6.3 Math 对象

在 JavaScript 中, 把所有复杂的数学公式与运算都封装在 Math 对象中。该对象不需要实例化就可以直接调用。例如, 求 6 除以 5 的整数值, 则可以使用如下方法来实现:

```
var n = Math.round(6 / 5); //调用 Math 对象的 round()方法
alert( n );               //返回 1
```


Math 对象属性主要是数学领域的专用值，常称之为数学常量，如表 17.5 所示。

表 17.5 Math 对象的常量

常 量	说 明
E	常量 e，自然对数的底数。例如， <code>alert(Math.E)</code> ，返回 2.718281828459045
LN10	10 的自然对数。例如， <code>alert(Math.LN10)</code> ，返回 2.302585092994046
LN2	2 的自然对数。例如， <code>alert(Math.LN2)</code> ，返回 0.6931471805599453
LOG10E	以 10 为底的 E 的对数。例如， <code>alert(Math.LOG10E)</code> ，返回 0.4342944819032518
LOG2E	以 2 为底的 E 的对数。例如， <code>alert(Math.LOG2E)</code> ，返回 1.4426950408889633
PI	圆周率圆的值。例如， <code>alert(Math.PI)</code> ，返回 3.141592653589793
SQRT1_2	2 的平方根除以 1。例如， <code>alert(Math.SQRT1_2)</code> ，返回 0.7071067811865476
SQRT2	2 的平方根。例如， <code>alert(Math.SQRT2)</code> ，返回 1.4142135623730951

Math 方法可以分为两类：专业计算方法（如表 17.6 所示）和数学常规方法（如表 17.7 所示）。

表 17.6 Math 专业方法

方 法	说 明
<code>sin()</code>	计算正弦值。例如， <code>alert(Math.sin(1))</code> ，返回 0.8414709848078965
<code>cos()</code>	计算余弦值。例如， <code>alert(Math.cos(1))</code> ，返回 0.5403023058681398
<code>tan()</code>	计算正切值。例如， <code>alert(Math.tan(1))</code> ，返回 1.5574077246549023
<code>atan()</code>	计算反正切值。例如， <code>alert(Math.atan(1))</code> ，返回 0.7853981633974483
<code>asin()</code>	计算反正弦值。例如， <code>alert(Math.asin(1))</code> ，返回 1.5707963267948965
<code>acos()</code>	计算反余弦值。例如， <code>alert(Math.acos(1))</code> ，返回 0
<code>atan2()</code>	计算从 X 轴到一个点的角度。例如， <code>alert(Math.atan2(50,50))</code> ，返回 0.7853981633974483。注意，是从 X 轴正向逆时针旋转到点 (x,y) 时经过的角度
<code>log()</code>	计算一个数的自然对数。例如， <code>alert(Math.log(1))</code> ，返回 0
<code>exp()</code>	计算 ex。例如， <code>alert(Math.exp(1))</code> ，返回 2.718281828459045
<code>pow(x,y)</code>	x 的 y 次幂，即 xy 。例如， <code>alert(Math.pow(3,4))</code> ，返回 81，等于 $3*3*3*3$
<code>sqrt()</code>	计算平方根。例如， <code>alert(Math.sqrt(4))</code> ，返回 2

表 17.7 Math 常规方法

方 法	说 明
<code>abs()</code>	计算绝对值。例如， <code>alert(Math.abs(-20))</code> ，返回数值 20
<code>round()</code>	舍入到最接近的整数。例如， <code>alert(Math.ceil(5.123))</code> ，返回 5；而 <code>alert(Math.round(-5.123))</code> ，返回 -5。注意，返回值是一个与参数最接近的整数
<code>ceil()</code>	对一个数上舍入。例如， <code>alert(Math.ceil(5.123))</code> ，返回 6；而 <code>alert(Math.ceil(-5.123))</code> ，返回 -5。注意，返回值是一个大于等于参数值，并且与它最接近的整数
<code>floor()</code>	对一个数下舍入。例如， <code>alert(Math.floor(5.123))</code> ，返回 5；而 <code>alert(Math.floor(-5.123))</code> ，返回 -6。注意，返回值是一个小于等于参数值，并且与它最接近的整数
<code>max()</code>	返回最大的参数。例如， <code>alert(Math.max(2,34,5,42))</code> ，返回 42
<code>min()</code>	返回最小的参数。例如， <code>alert(Math.min(2,34,5,42))</code> ，返回 2
<code>random()</code>	返回一个 0.0~1.0 之间的一个伪随机数。例如， <code>alert(Math.random())</code> ，返回 0.4449011053739194（每次都不同）

专业计算方法都是一些数学函数，如三角函数、指数对数计算、根幂计算等。在数学常规方法中，包括数值取值（如取整、取正）、随机数和数值比较。数值取值中的 `ceil()`、`floor()`、`round()` 3 个方法容易混淆，`min()` 和 `max()` 方法可以比较多个参数，并返回最小和最大值。`random()` 方法能够生成一个在 0~1 之间的随机数，不包括 0 和 1。如果希望获取指定范围的随机数，可以使用如下公式：

```
number = Math.random()*total + first
```

其中 `total` 表示随机数的范围，而 `first` 表示可能最小的随机数。例如，生成 10 个从 1~10 之间的随机数，则可以使用如下代码：

```
for( var i = 0; i < 10; i ++ ){
    //生成 1~10 之间的随机数（包括 1 和 10），其中 10 表示随机数的范围，1 表示随机数的起始值
    var n = Math.random() * 10 + 1;
    document.write( n + "<br />" );    //输出随机数
}
```

如果希望生成随机数为整数，则可以使用 `floor()` 进行取整，不可以使用 `ceil()`、`round()` 方法，因为它们会向上取值，导致超出指定范围。例如：

```
for( var i = 0; i < 10; i ++ ){
    var n = Math.floor(Math.random() * 10 + 1);
    document.write( n + "<br />" );
}
```

如果希望生成 10 个在 10~20 之间的随机整数，不包括 10 和 20，则可以使用如下代码：

```
for( var i = 0; i < 10; i ++ ){
    var n = Math.floor(Math.random() * 9 + 11);
    document.write( n + "<br />" );
}
```

其中数值 9 表示 10~20 之间的范围，而 11 表示随机数的可能最小值。

17.6.4 Date 对象

在 JavaScript 语言中，所有与时间有关的操作都由 `Date` 对象负责。而在一些特殊环境中，时间还可以作为数值类型来处理，如时间比较、时间运算。

创建时间对象的方法有如下 4 种。

1. 获取本地时间对象

```
var now = new Date();
alert(now); //返回当前时间对象，如 Wed Apr 29 15 :37: 55 UTC +0800 2009
```

2. 获取指定时间对象

构造函数 `Date()` 的参数格式如下：

```
new Date(year, month, day, hours, minutes, seconds, ms)
```

除了前两个参数（年和月）外，其他所有参数都是可选的。其中月数参数从 0 开始，如 0 表示第 1 个月，11 表示第 12 个月。例如：

```
var d1 = new Date(2009,4,1);
alert(d1); //返回时间对象，如 Fri May 1 00: 00:00 UTC+ 0800 2009
var d2 = new Date(2009,4,1,5,30,30);
alert(d2); //返回时间对象，如 Fri May 1 00: 00:00 UTC+ 0800 2009
```

所有声明的日期和时间使用的都是本地时间，而不是 UTC 时间（类似于 GMT 时间）。

3. 通过字符串创建时间对象

此时，月份是从 1 开始，而不是从 0 开始。例如：

```
var d1 = new Date("2009/4/1 5:30:30");
alert(d1); //返回时间对象, 如 Wed Apr 1 05 :30: 30 UTC +0800 2009
```

4. 通过毫秒数创建时间对象

这个毫秒数是距离 1970 年 1 月 1 日午夜(GMT 时间)的毫秒数。例如:

```
var d1 = new Date(1000000000000);
alert(d1); //返回时间对象, 如 Sun Sep 9 09 :46 :40 UTC +0800 2001
```

创建 Date 对象之后, 就可以调用该对象的各种方法操作时间了。Date 对象的方法包括两大类: 一类方法是设置时间, 如设置时间对象的小时字段 setHours(), 设置时间对象的月份字段 setMonth()等。另一类就是获取时间对象的各个字段值, 如获取时间对象的小时字段 getHours(), 获取时间对象的月份字段 getMonth()等。由于这些方法使用比较简单, 且用法类似, 这里就不再详细说明了, 读者可以参考 JavaScript 参考手册了解它们。

```
d = new Date( ); //获取当前日期和时间
alert(d.toLocaleDateString()); //显示日期
alert(d.toLocaleTimeString()); //显示时间
alert(d.getDay()); //获取一周中的第几天
```

利用 Date 对象还可以判断两个时间的时差。例如, 下面的示例可以计算一个循环体空转 100 万次所花费的毫秒数:

```
var d1 = new Date();
var i = 0;
while(true){
    i ++;
    if(i > 1000000)
        break;
}
var d2 = new Date();
alert(d2 - d1); //返回循环体运行的时间
```

17.7 类 型

类的概念源于人们认识自然、认识社会的过程。在这一过程中, 主要借助两种方法来实现: 归纳法和演绎法。

- ☑ 所谓归纳法, 就是由特殊到一般。在归纳的过程中, 从具体的事物中把共同的特征抽取出来, 形成一般的概念, 这就是归类。例如, 壁虎、狮子、鲫鱼、麻雀, 因为它们都能够自由活动, 所以被归类为动物; 而松树、小草、浮萍、苔藓, 因为它们都不能够自由活动, 所以被归类为植物。
- ☑ 所谓演绎法, 就是由一般到特殊。在演绎的过程中, 人们把同类的事物, 根据不同的特征分成不同的小类, 这就是分类。例如, 动物可以细分为猫科动物, 猫科动物可以细分为猫, 而猫又细分为花猫、白猫等。

对于具体的类, 它会包括很多具体的个体, 这些个体就是对象。类的内部状态是指类集合中对象的共同状态, 类的运动规律是指类集合中对象的共同运动规律。例如, 人是能够思维的高级动物, 它被称为类, 它的上面还包含父类(专业称为超类, 如动物)、祖类(专业称为基类, 如细菌)等, 而张三、李四、王五就是具体的人, 这些具体的人就是对象。人与人生活在一起就构成了社会(即作用域), 社会中各种交往活动就构成了人的生活规律。

17.7.1 认识类

类的本质是抽象, 抽象是一个分析的过程。抽象的过程就是定义类知道和要完成的事情的过程。因此,

它具有 3 个基本特性，即继承、封装和多态。

- ☑ 继承：描述了类型的血缘关系，不同类之间经常会存在相似性。两个以上的类也会经常共享相同的属性或方法。因为我们并不想重写代码，于是就利用继承机制来快速实现代码的“复制和粘贴”。当然继承机制比较复杂，这决定了它不是一项简单的手工劳动。
- ☑ 封装：描述了类型的交流方式。一面之交，也许我们并不知道类型的内部机制和实现，当然也不想知道，我们只知道如何使用类型即可。封装暗示着我们能够以任何方式构建系统，如果需要，还可以在日后再次修改其内部的结构，只要系统中不同功能组件之间的接口没有发生变化，那么对系统中一个功能部分的改变不会对系统的其他功能部分产生影响。
- ☑ 多态：描述了类型的宽容性。通过多态，可以在事先不知道对象的类型时就与它进行协作，即使类型不同，而类能够帮助我们处理这些问题。

类还包括很多成员，类成员描述了类内部的各种各样的事物，不同语言都有自己的规定，如常量、字段、构造器、方法、属性、事件、操作符、重载、类型等。JavaScript 是一种弱类型语言，它没有那么多约定，只定义了基本成员，如属性、方法和事件。成员的名称也被称为标识符。

属性（Property）是数据，而方法（Method）是函数。属性是类知道的事情，而方法是类准备完成的事情。属性和方法都是类的核心。在很多场合我们也习惯于把它们都视为属性，称之为对象的属性，此时属性实际上应该算是特性（Attribute）。

面向对象开发是基于这样的概念：系统应由对象来创建，对象拥有数据和功能。属性定义数据，而方法定义功能。

显然，在面向对象的开发中，最重要的工作就是创建类。而创建类时，就必须定义它的属性和方法。属性的定义应该是直接明了的，需要定义它的名称和数据类型。方法的定义就是创建一个函数的过程，根据需要，还可以创建能够接收参数且能够返回值的方法。

在面向对象的程序设计中，所有行为都是以事件驱动来实现的，这意味着假如没有任何事件发生，程序仅仅是多行无意义的文本字符串。

17.7.2 定义类

Object、Function、Array 等对象都是抽象类，或称之为构造函数，当然内置对象仅是 JavaScript 支持面向对象编程的一部分，JavaScript 支持用户自定义类。

1. 工厂模式

简单的工厂模式代码如下：

```
function wrap(){           //使用函数结构体包装对象
    var book = new Object();
    book.title = "JavaScript 类";
    book.pages = 200;
    book.what = function(){
        alert(this.title +this.pages)
    }
    return book;           //返回初始化后的对象
}
var book1 = wrap();         //调用模具快速生产产品 1
var book2 = wrap();         //调用模具快速生产产品 2
var book3 = wrap();         //调用模具快速生产产品 3
```

带有参数的工厂模式定义类如下：

```
function wrap(title,pages){ //通过包装函数传递属性的默认值
    var book = new Object();
    book.title = title;      //声明属性并接收参数值
```

```

    book.pages = pages;           //声明属性并接收参数值
    book.what = function(){
        alert(this.title + this.pages)
    }
    return book;                 //返回初始化后的对象
}
var book1 = wrap("小李",160);    //调用模具快速生产，生产时快速设计书名
var book2 = wrap("老赵",240);    //调用模具快速生产，根据读者水平调整书的容量
分离方法的工厂模式如下：
what = function(){              //声明一个全局方法
    alert(this.title + this.pages)
}
function wrap(title,pages){
    var book = new Object();
    book.title = title;
    book.pages = pages;
    book.what = what;            //引用全局方法的引用指针
    return book;
}
var book1 = wrap("小李",160);
var book2 = wrap("老赵",240);

```

通过分离方法，从而改进了重复创建相同函数的弊端，在模具中直接引用方法的指针，实例直接调用公共方法，节省了大量资源。

在应用工厂模具时，使用 `new` 运算符来调用，使其看起来像真正的构造函数。代码如下：

```

var book1 = new wrap("小李",160);
var book2 = new wrap("老赵",240);

```

由于在函数 `wrap()` 内部已经调用了 `new` 运算符，所以模具（伪造构造函数）外面这个 `new` 运算符就被忽略。在函数 `wrap()` 内部创建的对象被返回并赋值给变量 `book1` 和 `book2`。

2. 构造模式

构造模式代码如下：

```

function Book(title,pages){      //构造函数模型
    this.title = title;          //构造函数的属性
    this.pages = pages;          //构造函数的属性
    this.what = function(){      //构造函数的方法
        alert(this.title + this.pages)
    };
}
var book1 = new Book("小李",160); //实例化构造函数，并传递参数
var book2 = new Book("老赵",240); //实例化构造函数，并传递参数

```

在 JavaScript 中，`Object`、`Array`、`Function`、`RegExp`、`String` 等内置对象都是构造函数，使用 `new` 运算符可以把这些预定义的构造函数实例化为对象。在 JavaScript 中，构造函数是具有如下特性的普通函数：

- ☒ 构造函数只能够使用 `new` 运算符进行调用，不能使用小括号调用。
- ☒ 构造函数内部通过 `this` 关键字指代当前对象自身。
- ☒ 构造函数内部必须通过点运算符来声明和引用成员。当然，构造函数结构体内也可以包含一般函数的执行语句。
- ☒ 传递给构造函数的是一个对新创建的空对象的引用，将该引用作为关键字 `this` 的值，而且构造函数还要对新创建的对象进行适当的初始化。

例如，下面这个构造函数 `Box()`，传递给 `Box()` 的是一个新创建的空对象，是一个高度抽象但未知的对

象, 通过 `this` 关键字来代称, 即 `this` 的值就是这个新创建的空对象引用。当使用 `new` 运算符实例化构造函数时, 可以通过传递参数来初始化这个对象的属性值。

```
function Box(w,h){           //构造函数
    this.w = w;              //构造函数的成员
    this.h = h;              //构造函数的成员
}
var box1 = new Box(4,5);     //实例并初始化构造函数
```

由于每一个构造函数都定义了对象的一个类, 所以给每个构造函数一个名字以说明它所创建的对象类就显得比较重要了, 类名应该很直观, 且首字母要大写 (非强制的), 主要是与普通函数进行区别。

构造函数只是初始化了的特定对象, 但并不返回这个对象。如果通过构造函数的结构来定义了一个对象类, 现在可以确保所有由 `Box()` 构造函数创建的对象都有初始化的 `w` 和 `h` 属性。这意味着可以在此基础上编写程序, 统一处理所有的 `Box` 对象。

构造函数没有返回值, 这是它与普通函数的一个最大区别。它们只是初始化由 `this` 关键字传递进来的对象。但是, 构造函数可以返回一个对象值, 如果这样做, 被返回的对象就成了 `new` 表达式的值。在这种情况下, `this` 值所引用的对象就被丢弃了。例如:

```
function Box(w,h){           //构造函数
    this.w = w;
    this.h = h;
    return this;              //返回关键字 this
}
alert(Box(4,5).w);           //返回参数值 4, 此时构造函数成为普通的函数
//返回参数值 4, 此时构造函数成为普通的函数
不过依然可以实例化构造函数, 并调用该对象的属性。例如:
var box1 = new Box(4,5);     //实例并初始化构造函数
alert(box1.w);                //调用对象的属性
```

通过上面示例可以看到, 构造函数内部没有创建对象, 而是使用关键字 `this` 来含糊地代替。当使用 `new` 运算符调用构造函数时, JavaScript 会自动创建一个空白的对象, 然后把这个空对象传递给 `this` 关键字, 作为它的引用值。这样 `this` 就成为新创建对象的引用指针了。

如同工厂函数一样, 构造函数会重复复制相同的函数 `what()`, 为每个对象实例都创建了独立的函数 `what()`。从效率的角度来说, 它只不过利用构造函数的形式进行封装会优化执行的流程, 间接提升了一点效率。不过可以把其中的函数 `what()` 移植到构造函数的外面, 然后通过引用的方式为构造函数的属性进行赋值。

3. 原型模式

在构造函数中定义 `prototype` 属性之后, 则任何实例对象都将拥有 `prototype` 属性所定义的属性。

例如, 先声明一个空构造函数, 并利用构造函数的 `prototype` 属性为该构造函数定义原型属性 `title` 和 `pages`, 以及原型方法 `what()`。构造函数的原型成员将会被所有实例对象继承。这样当使用 `new` 运算符实例化对象时, 所有对象都拥有原型属性中定义的成员。

```
function Book(){              //使用构造函数定义一个空类
}
Book.prototype.title = "JavaScript 类"; //为原型声明一个属性并初始化
Book.prototype.pages = 200;           //为原型声明一个属性并初始化
Book.prototype.what = function(){      //为原型声明一个方法
    alert(this.title + this.pages);
};
var book1 = new Book();              //实例化对象 book1
var book2 = new Book();              //实例化对象 book2
var book3 = new Book();              //实例化对象 book2
alert(book1.title);                  //返回字符串 JavaScript 类
```

```

alert(book2.pages);           //返回值 200
book3.what();                 //调用对象的方法 what()

```

如果使用 `instanceof` 运算符检测对象实例的类型，可以发现下面的表达式返回值为 `true`，说明 `book1` 变量的类型为 `Book`：

```

alert( book1 instanceof Book); //返回 true

```

属性 `prototype` 是在函数作为构造函数时使用的。例如，下面的用法是错误的，因为对象 `o` 是一个实例，它无权为自己定义原型：

```

var o =new Object()           //实例化对象
o.prototype.title = "JavaScript 类"; //试图为该对象定义原型属性
alert(o.title);               //返回错误信息

```

但是下面的用法是正确的：

```

Object.prototype.title = "JavaScript 类"; //为 Object 对象类声明原型属性，并初始化
var o = new Object();                     //实例化对象
alert(o.title);                           //继承属性

```

这是因为 JavaScript 内置对象都是构造函数结构体，因此它们都拥有 `prototype` 属性，并利用它为自己定义原型属性或原型方法，从而实现对内置对象进行功能扩展。

4. 构造原型模式

原型模式存在以下两个问题：

- ☑ 由于构造函数事先被声明，而原型属性在类结构声明之后才被定义，因此无法通过构造函数参数向原型属性动态传递值。由该类实例化的所有对象都是一个模样，要改变原型属性值，则所有实例都受到干扰。
- ☑ 当原型属性的值为引用类型数据时，如果在一个对象实例中修改该属性值，将会影响所有的实例，而不仅仅是它自己的。

```

function Book(){               //声明构造函数
}
Book.prototype.o = {x:1,y:2}   //构造函数的原型属性 o 是一个对象
var book1 = new Book();        //实例化对象 book1
var book2 = new Book();        //实例化对象 book2
alert(book1.o.x);              //返回 1
alert(book2.o.x);              //返回 1
book2.o.x = 3;                 //修改实例化对象 book2 中的属性 x 的值
alert(book1.o.x);              //返回 3
alert(book2.o.x);              //返回 3

```

由于原型属性 `x` 的值为一个引用类型数据，因此所有对象实例的属性 `x` 的值都是指向该对象的引用指针。因此一旦某个对象的属性值被改动，其他实例对象的属性值也会随之发生变化。

而构造函数原型模式正是为了解决原型模式而诞生的一种混合设计模式，它是把构造函数模式与原型模式混合使用，从而避免了此类问题的发生。具体的方法为：对于可能会相互影响的原型属性，并且希望动态传递参数的属性，则拆分出来使用构造函数模式进行设计。而对于不需要个性，反而希望共享，且又不会相互影响的方法或属性，则单独使用原型模式来设计。

遵循这样的设计原则，对于上面示例中存在的问题，可以把其中的两个属性设计为构造函数模式，而对象方法设计为原型模式。具体代码如下：

```

function Book(title,pages){    //构造函数模式设计
    this.title = title;
    this.pages = pages;
}
Book.prototype.what = function(){ //原型模式设计

```

```

    alert(this.title + this.pages);
};
var book1 = new Book("小李",160);
var book2 = new Book("老赵",240);
alert(book1.title);
alert(book2.title);

```

一般在混合使用构造函数与原型模式时，可以不使用构造函数定义对象的所有非函数属性（即对象属性），而使用原型模式定义对象的函数属性（即对象方法）。这样所有方法都只创建一次，而每个对象都能够根据需要自定义属性值。

这种混合型杂交模式成为 ECMAScript 定义类的推荐标准，这也是使用最广的一种设计模式，它具有前面设计模式的所有优点。

5. 动态原型模式

从面向对象的设计原则来思考，所有成员都被封装在类结构体内，因此可以这样完善上面示例的设计思路：

```

function Book(title,pages){                //构造函数模式设计
    this.title = title;
    this.pages = pages;
    Book.prototype.what = function(){      //原型模式设计，位于类的内部
        alert(this.title + this.pages);
    };
}
var book1 = new Book("小李",160);
var book2 = new Book("老赵",240);
alert(book1.title);
alert(book2.title);

```

当每次实例化对象时，类 Book 中包含的原型方法就会被创建一次，这里不妨使用分支结构封装该原型方法，判断如果原型方法存在，则就不再创建该方法，否则就创建方法。代码如下：

```

function Book(title,pages){
    this.title = title;
    this.pages = pages;
    if(typeof Book.isLock == "undefined"){ //创建原型方法的锁，如果不存在该方法则创建
        Book.prototype.what = function(){
            alert(this.title + this.pages);
        };
        Book.isLock = true;                //创建原型方法后，把锁锁上，避免重复创建
    }
}
var book1 = new Book("小李",160);
var book2 = new Book("老赵",240);
alert(book1.title);
alert(book2.title);

```

`typeof Book.isLock` 表达式能够检测该属性值的类型，如果返回为 `undefined` 字符串，则不存在该属性值，即说明还没有创建原型方法，则允许进入分支结构创建原型方法，并设置该属性的值为 `true`，这样它的类型返回值就是 `boolean` 字符串了，从而间接锁定了通道，不允许逻辑再次进入分支。这里使用类名 `Book`，而没有使用 `this` 关键字，这是因为原型是属于类本身的，而不是对象实例的。

动态原型模式与构造函数原型模式在性能上是等价的，读者可以自由选择这两种方式，不过目前使用最广泛的是构造函数原型混合模式。

17.8 接 口

接口是面向对象编程的基础，它是一组包含了函数型方法的数据结构，与类一样都是编程语言中比较抽象的概念。联系生活中的各种接口，如利用机顶盒这个接口，能够收看到不同频道和信号的节目，它犹如对不同类型的信息进行集合和封装的设备，最后把各种不同类型的信息转换为电视能够识别的信息。对于编程语言中的接口，它实际上是不同类的封装，并提供统一的外部联系通道，这样其他对象就可以利用接口来调用不同类的成员了（如属性和方法）。

17.8.1 认识接口

构造函数（类）是具体实现，接口是类的约定。API 接口（应用程序接口）、人机交互接口、电源接口、USB 接口等虽然用途不同，功能各异，但是都包含一个共同的特性：约定、规范。可以说，接口就是一张契约或合同，它约定了设计者和使用者都必须遵循的要求。

接口（Interface）承诺了具体类应该实现的功能。在 Java 或 C# 语言中，Interface 结构里的函数声明就是这种承诺。例如，Base 承诺了 3 个基本功能：function1()、function2() 和 function3()，则 Java 可以这样书写：

```
interface Base{           //接口，约定 3 个基本功能
    void function1();
    void function2();
    void function3();
}
```

这份合同在甲方、乙方之间签订：乙方是定义类的开发人员，甲方是调用类的用户。

开发人员负责实现接口约定的功能。实现在编程语言里使用 implements 关键字来表示，功能的实现者就是所谓的类（即 Class）。

乙方在 Java 中可以这样写：

```
class App implements Base //定义类，接口实现
```

即类 App 将遵循 Base 接口约定，从专业术语角度来说，就是应用类 App 继承于 Base 接口类。3 个功能 function1()、function2()、function3() 的具体实现代码也要放在类 App 中。具体实现如下：

```
class App implements Base{ //定义类，实现 3 个基本功能
    void function1(){
        System.out.println( "function1" );
    }
    void function2(){
        System.out.println( "function2" );
    }
    void function3(){
        System.out.println( "function3" );
    }
}
```

乙方实现了对合同的承诺。当然，甲方也必须兑现合同的承诺。如何兑现呢？甲方不清楚乙方的技术实现细节，当然也没有这个必要，更不会关心甲方是如何实现这些约定的。甲方只需要根据合同说明，遵循合同条款正确使用类即可。于是，接口就成为甲方和乙方遵循相同规则的纽带。

当然，回到程序设计上来，接口（interface）和类（class）实际上都是相同的数据结构。接口中可以声明属性、方法、事件和类型，但不能声明变量，且不能设置被声明成员的具体值（功能实现）。也就是说，接口只能定义成员，不能给定义的成员赋值。而接口作为它的继承类或者派生类的契约，继承类或者它的

派生类应共同完成接口属性、方法、事件和类型的具体实现。在接口与实现类之间，不管是方法名还是属性调用顺序上都应保持一致的。

接口的目的就是要约束编码，促使代码规范。这对于强类型的语言来说是必需的，也是非常重要的环节。但是对于 JavaScript 这类弱类型的语言，严格的类型检查往往会束缚 JavaScript 的灵活性。很多前端开发人员根本就不用接口，但是不会影响 JavaScript 脚本的设计。

使用接口可以降低对象间的耦合度，提高代码的灵活性。学会使用接口，能够让手中的函数变得灵巧而又乖巧，这在大型开发中是非常重要的。

试想一下，在一个大型项目中，会有无数的功能块以及外部 API。很多功能块或 API 可能还没有开发出来，但是在项目部署中又必须考虑这些问题。这时候，如果定义一个或多个接口，让接口约定功能块和 API 的实现，然后有时间时，或者其他开发人员就可以根据这个接口来开发具体的实现功能块。

也许那位开发人员实现指定功能的具体途径不同，但是他遵循接口约定，即可实现即插即用，不会担心自己开发的功能块与预留的占位代码不相容。

当然，针对 JavaScript 来说，使用接口还存在很多好处。例如，接口能够实现类之间的通信，不管类之间的功能如何迥异，只要它们拥有相同的接口，类之间的交流就不成问题。另外，使用接口能够帮助监测代码运行，这使 JavaScript 调试变得方便很多。

JavaScript 语言自身的特性决定了接口鸡肋成分大于鸡翅的因素。首先，JavaScript 是弱类型语言，强制性显得不是那么必要。其次，JavaScript 不支持接口功能，没有提供内置方法。如果人工地设计一个额外的接口程序，这将对程序的性能产生一定的影响。项目越大，这种开销可能也就越大。最后，JavaScript 中的接口仅是一种期望，而不是强迫，这与 Java 或 C# 等强类型语言不同。在 JavaScript 中，则更多地体现在开发人员的自觉行为，这在一定程度上减少了接口的实用价值。

到底用不用接口？这个问题确实很困惑人，可以遵循下面条件作为选用标准：

- ☒ 项目的大小。如果是一个框架，使用接口能够提升程序的性能。如果是一些简单的应用，使用接口就有点费力不讨好了。
- ☒ 量力而行。如果驾驭 JavaScript 接口比较娴熟，那么多用接口也不是坏事。如果担心接口额外影响程序的性能，则可以考虑在发布产品时，清除接口功能模块，或者设置接口执行条件，防止它被频繁执行，这样也没有必要。

17.8.2 定义接口

JavaScript 不支持接口，但是可以模仿其他语言形式定义接口。首先，构建一个简单的类：

```
function Neddy(){
    this.name = undefined;
}
Neddy.prototype.set = function(name){
    this.name = name;
}
Neddy.prototype.get = function(){
    alert(this.name);
}
```

//创建木驴基本构造
//安装一个大脑，能够存储信息
//绑定一个眼睛，能够接收操纵者的眼色
//绑定一个神经，能够把大脑存储信息传输出去

然后仔细分析并提炼，概括出木驴结构的基本行为标准：

```
/*
//接口结构，标准木驴结构约定的基本行为方法
interface Base{
    function set(string);
    function get();
}
```

//能够观察操纵者的眼色
//能够根据大脑中存储的操纵者的眼色行事

```

}
*/

```

设计接口监控的框架函数：

```

function interface(o){           //接口监控函数
    if(!implements(o,"set","get")){ //如果不符合实现约定，则抛出一个异常
        throw new Error("类没有实现接口约定");
    }
}

```

在上面的函数中，参数 *o* 表示类的实例。根据标准约定，类中必须定义 *set* 和 *get* 方法。而 *implements()* 函数就是具体检测的实验员了。它将根据类提供的样品标准与接口约定中（即第 2、3 个参数值）的标准进行比对。如果相同则放行；如果不同，说明类不符合接口约定，就不放行，并抛出一个错误。

有了基本的思路，剩下的问题就是技术实现问题了。代码如下：

```

function implements(o){           //接口标准的化学检测函数
    for(var i = 1;i<arguments.length;i++){ //遍历接口约定的标准，在第 2、3 个参数中输入
        var p = arguments[i];           //把接口约定标准放入试管
        var b = false;                   //声明一个临时记录器
        for(var j=0;j<o.base.length;j++){ //遍历类提供的实现标准，类提供的样品以数组形式存储在 base 属性中
            if(o.base[j] == p) b = true; //如果比对一致，则在临时记录器中做一个对号
        }
        if(!b) return false; //如果发现记录器中没有一个对号，说明类提供样品标准与接口标准不同，则退出函数，
            返回 false
    }
    return true;                       //完全检测通过，则返回 true
}

```

实现接口检测：

```

function Neddy(){                 //完善木驴标准结构
    this.name = undefined;
    this.base = ["set","get"];    //以数组形式存放本类实现的方法
    interface(this);              //调用接口检测工具进行检测
}

```

调试接口：

```

var neddy = new Neddy();          //实例化样品
neddy.set("Ghost");               //发出指令进行调试
neddy.get();                       //正确执行

```

上面示例对类所实现的接口提供了说明，并适当进行约束。例如，下面不按规范操作，接口会提示错误信息：

```

function Neddy(){                 //制作一匹懒驴
    this.name = undefined;
    this.base = ["set"];          //驴的功能标准
    interface(this);              //接口标准监测
}
Neddy.prototype.set = function(name){ //只听人话，不做人事
    this.name = name;
}

```

上面实现接口的方法也存在以下一些问题：

- ☒ 监督方法是伪监督，没有透明度。试想一下，如果用户知道了接口标准，仅定义了一个方法，但是却在 *base* 属性中说自己定义了两个方法：

```

function Neddy(){                 //制作一匹懒驴
    this.name = undefined;

```

```

is.base = ["set", "get"];           //驴的功能标准
interface(this);                   //接口标准监测
}
Neddy.prototype.set = function(name){ //只听人话，不做人
    this.name = name;
}

```

- ☒ 监督失去了公平性。本来接口约定的是甲乙双方的行为，在 Java 和 C#等语言中，如果接口约定了类方法的参数类型或返回值类型，那么使用者必须按规定输入指定类型的值，否则是错误的。但是这种方法仅对乙方有效，却不管甲方在使用时的行为。

第一个问题：如果要想类的成员如实与接口标准一致，就不能够听从类自说自演，必须实地考察才是。假设从接口标准中获取方法集，然后循环检测每个方法名在类中是否已经实现。核心表达式如下：

```
o[method]
```

其中 o 表示接口实现的类，method 表示接口标准中指定的方法名。如果上面表达式为 true，则说明实现的类定义了约定的方法。

但是由于类的成员不仅仅包括方法，还可能包含属性，因此还要检测类中的 method 名称为函数类型，即核心表达式为：

```
typeof o[method] == 'function'
```

第二个问题：这里假设要检测甲方输入参数时必须遵循的约定。也就是说，继续检测类方法的参数个数与接口标准中指定的参数是否一致，则核心表达式为：

```
o[method].length== interface.methods[j][1]
```

在上面的表达式中，o[method].length 子表达式表示实现类的方法参数个数，右侧表达式 interface.methods[j][1]表示接口标准中约定的指定方法的参数个数。

核心的问题有了解决思路，下面来规划接口结构的设计和检测功能的实现。具体操作步骤如下。

(1) 设计一个接口辅助的类结构。这个构造函数相当于过滤器，用来在接口实例化过程中监测初始化参数是否合法。如果符合接口设计标准，则把第 2 个参数中每个方法名和参数个数以数组元素的形式输入接口内部属性 methods。在输入前分别检测每个参数的类型是否符合规定，同时检查参数是否存在残缺，并及时以 0 补齐参数。代码如下：

```

function Interface( name, methods ){           //接口辅助类，参数包括接口实例的名称和方法集
    if( arguments.length != 2 ){               //如果参数个数不等于 2，则抛出错误
        throw new Error( "该接口实例包含" + arguments.length + "个参数，但标准接口约定两个参数." );
    }
    this.name = name;                          //存储第 1 个参数值
    this.methods = [];                         //接口实例的方法存储器
    if( methods.length < 1 ){                  //如果第 2 个参数元素数为 0，说明是空数组，抛出异常
        throw new Error( "接口实例的第 2 个参数是空的。" );
    }
    for( var i = 0; i < methods.length; i ++ ){ //遍历第 2 个参数的每个元素，合乎接口标准设计要求，则写入接口实例的方法存储器
        this.methods
            if( typeof methods[i][0] !== 'string' ){
                throw new Error( "接口约定的第 1 个参数应为字符串。" );
            }
            if( methods[i][1] && typeof methods[i][1] !== 'number' ){
                throw new Error( "接口约定的第 2 个参数应为数值。" );
            }
            if( methods[i].length == 1 ){
                methods[i][1] = 0;
            }
        }
    }
}

```

```

    }
    this.methods.push( methods[i] );           //把方法写入数组存储器
  }
}

```

(2) 为接口辅助类 Interface 定义一个方法 implements, 该方法将检测实现类 (即继承该接口实例标准的构造函数) 是否符合接口实例的约定。它至少包含两个参数, 第 1 个参数 o 表示实现类, 第 2 个参数及其后面的参数表示该类将要实现的接口标准。也就是说, 可以为一个类指定多个接口约定, 这样就能够更灵活地分类设计接口实例。

然后遍历第 2 个及其后面所有参数, 在循环结构中, 先检测接口是否为接口标准的实例, 否则就会抛出异常。再从接口实例的方法存储器中逐一读取方法名, 填入类中来验证类的方法是否符合接口实例设置的标准, 验证包括方法名、function 类型和参数个数。如果发现问题, 则会立即抛出异常。代码如下:

```

Interface.implements = function( o ){           //接口辅助类, 检测类方法是否符合接口实例约定
  if( arguments.length < 2 ){                 //检测该方法的传递参数值是否符合规定
    throw new Error( "接口约定类应包含至少两个参数。" );
  }
  for( var i = 1; i < arguments.length; i ++ ){ //检测类所遵守接口实例是否合法 (符合接口标准设计)
    var interface = arguments[i];
    if( interface.constructor !== Interface ){ //检测接口实例是否继承接口标准
      throw new Error( "从第 2 个以上的参数必须为接口实例。" );
    }
    for( var j = 0; j < interface.methods.length; j ++ ){ //检测类方法是否符合接口实例的约定
      var method = interface.methods[j][0];
      if( ! o[method] || typeof o[method] !== 'function' ||
        o[method].length !== interface.methods[j][1] ){
        throw new Error( "该实现类没能履行" + interface.name +
          "接口方法" + method + "约定。" );
      }
    }
  }
}

```

(3) 实例化接口标准。Interface 接口仅是一个构造函数, 也就是一个框架协议, 还没有指定类应该遵循的具体标准。框架协议中已经设计好了监测逻辑, 一旦实例化接口, 并指明类应遵守的约定, 那么应用该标准实例的类就必须遵守。为了帮助读者更清楚地理解接口标准 (接口构造函数、接口类)、接口实例的关系, 这里根据 Interface 接口标准定义了 6 个具体的接口实例:

```

var Get = new Interface( "Get", [ "get", 0 ] ); //该标准包含方法 get()
var Set = new Interface( "Set", [ "set", 1 ] ); //该标准包含方法 set()
var Saying = new Interface( "Saying", [ "saying", 1 ] ); //该标准包含方法 saying()
var Base = new Interface( "Base", [ "get", 0, "set", 1 ] ); //该标准包含方法 get()和 set()
var Base1 = new Interface( "Base1", [ "set", 1, "get", 0 ] ); //该标准包含方法 set()和 get()
var Base2 = new Interface( "Base2", [ "get", 0, "set", 1, "saying", 1 ] ); //该标准包含 3 个方法

```

(4) 在类中定义应该实现的标准, 即类应该遵循的接口约定。代码如下:

```

function Neddy(){                               //接口实现类
  this.name = "";
  Interface.implements( this, Base, Get, Set ); //绑定应该实现的接口实例, 即引用接口标准的 implements()方法, 并指定要遵守的接口实例约定, 在这里可以指定多个, 也可以仅指定一个
}

```



```

Neddy.prototype.get = function(){
    return this.name;
}
Neddy.prototype.set = function( name ){
    this.name = name;
}

```

(5) 在类中设置多个接口实例，但要确保类中定义了接口实例中约定的方法，否则就会报错。上面代码中为 Neddy 绑定了 3 个接口实例，由于类中所定义的方法都符合它们的约定，所以可以正常执行。但是如果指定其他 3 个接口实例中的一个或多个，就会抛出异常。这是因为当前类 Neddy 还没有实现它们的约定，即没有定义方法 saying()。

```

var neddy = new Neddy();
neddy.set( "Testing..." );
alert( neddy.get() );

```

//返回字符串 Testing..., 接口约定顺利通过

17.9 原型

原型是 JavaScript 语言的核心技术之一，它通过 prototype 这个属性表现出来。在 JavaScript 中，对象（Object）是没有原型的，仅构造函数拥有原型，而构造类的实例对象都能够通过 prototype 属性访问或操作原型。

17.9.1 认识 prototype

prototype 是对象类的原型，实现和管理继承的一种机制。prototype 属性是在函数作为构造函数时使用的。它引用的是作为构造函数的原型的对象。由这个构造函数创建的任何对象都会继承属性 prototype 引用的对象的所有属性。

从语义角度分析，prototype 表示类的原型，就是构造类拥有的原始成员（如属性和方法等）。

从语法角度分析，prototype 是构造函数的一个专有属性，该属性仅供构造函数类使用，其他类或对象都无权调用。prototype 属性存储着一个引用对象的指针值，该指针指向一个原型对象，它是一个特殊的对象，相当于一个数据集合，内部存储着构造函数的原始模型，即原型属性和方法。

当构造函数实例化后，所有被创建的实例对象都将拥有属性 prototype，它们都指向同一个引用对象，即原型对象。因此，构造函数的实例都可以访问构造函数的原型数据。如果在原型对象中声明一个成员，则所有对象实例都可以共享，它相当于一个中央仓库，完全面向构造函数类的所有子类或实例公开，而不是仅供构造函数自身享用。

原型实际上也是一个对象，继承于 Object 抽象类，不过它比较特殊，由 JavaScript 自动创建并依附于每个构造函数类上，从而实现构造类的原型属性和原型方法能够被所有实例对象继承。原型在 JavaScript 对象系统中的位置和关系如图 17.2 所示。

在 JavaScript 中，对象应该是类（Class）和实例（Instance）的关系演化，也就是说，是对象类创建对象实例。类是对象的模型化，而实例则是类的特征具体化，或者说类是实例的类型（type）。类又包含很多概念类型，如元类、超类、泛类和类型等，这些概念的不同，无非说明类的抽象程度不同。

一般所说的对象就是对象实例，而不是对象类。不过在 JavaScript 中，不支持类这个概念，类是通过构造函数来实现的类模块。例如：

```

function Class(type){
    this.type = type;
}

```

//构造函数，使用构造器构造类

```

}
var instance1 = new Class("instance1");    //实例对象 1
var instance2 = new Class("instance2");    //实例对象 2

```

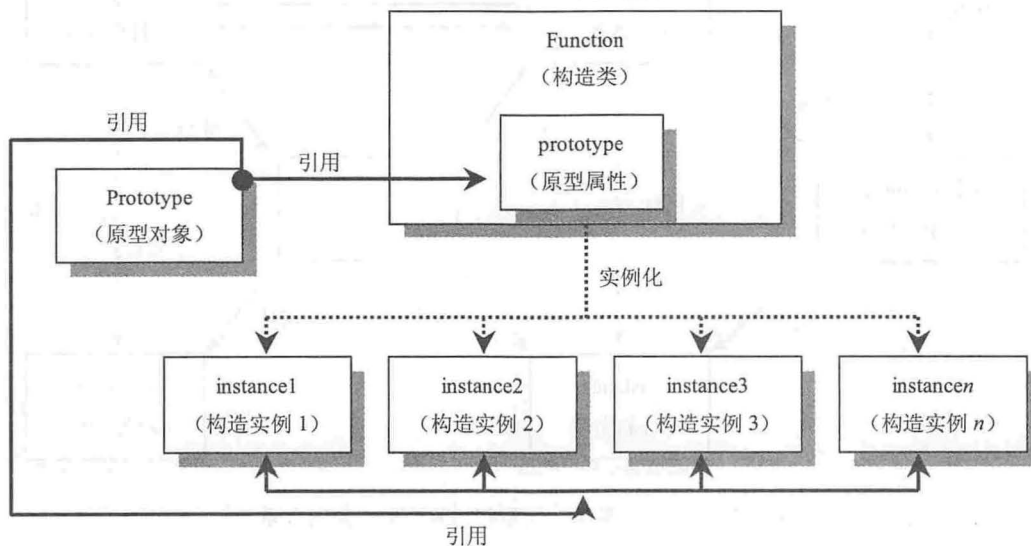


图 17.2 原型对象、原型属性及在对象系统中的位置关系

如果使用 instanceof 运算符可以验证它们的关系：

```

alert(instance1 instanceof Class);    //返回 true, 说明 instance1 对象是 Class 构造函数的实例
alert(instance2 instanceof Class);    //返回 true, 说明 instance2 对象是 Class 构造函数的实例

```

instance1 和 instance2 都是对象，但是 Class 构造函数不是它们唯一的类型，Object 也是它们的类型。例如：

```

alert(instance1 instanceof Object);    //返回 true, 说明 instance1 对象也是 Object 构造函数的实例
alert(instance2 instanceof Object);    //返回 true, 说明 instance2 对象也是 Object 构造函数的实例

```

因为 Object 对象比 Class 类型更加抽象，它们之间应该属于一种衍生关系，通俗地说就是继承关系：

```

alert(Class instanceof Object);    //返回 true, 说明 Class 类是 Object 对象的实例（或子类）

```

但是 instance1 和 instance2 对象却不是 Function 构造函数的实例，说明它们之间没有直接关系：

```

alert(instance1 instanceof Function);    //返回 false, 说明它们不是类型与实例的关系
alert(instance2 instanceof Function);    //返回 false, 说明它们不是类型与实例的关系

```

而 Object 与 Function 之间的关系就非常微妙，它们都是高度抽象的类型，相互之间都是对方的实例：

```

alert(Object instanceof Function);    //返回 true, 说明 Object 对象是 Function 函数的实例（即子类）
alert(Function instanceof Object);    //返回 true, 说明 Function 函数是 Object 对象的实例（即子类）

```

Object 与 Function 同时也是两个不同类型的构造器，利用运算符 new，可以设计不同性质的对象实例。

下面的代码能够很好地显示它们之间的差异：

```

var f = new Function();    //实例化 Function 对象
var o = new Object();    //实例化 Object 对象
alert(f instanceof Function);    //返回 true, 说明 f 是 Function 对象的实例
alert(f instanceof Object);    //返回 true, 说明 f 是 Object 对象的实例
alert(o instanceof Function);    //返回 false, 说明 o 不是 Function 对象的实例
alert(o instanceof Object);    //返回 true, 说明 o 是 Object 对象的实例

```

下面两行代码能够说明它们实例化对象之后的差异：

```

f.prototype.a = true;    //为实例 f 设置 prototype 原型属性，操作成功，说明 Function 对象拥有原型属性
o.prototype.a = true;    //为实例 o 设置 prototype 原型属性，操作失败，说明 Object 对象不拥有原型属性

```

instance（实例对象）、Class（类型）、Object（抽象类）和 Function（构造类）之间的关系如图 17.3 所示。

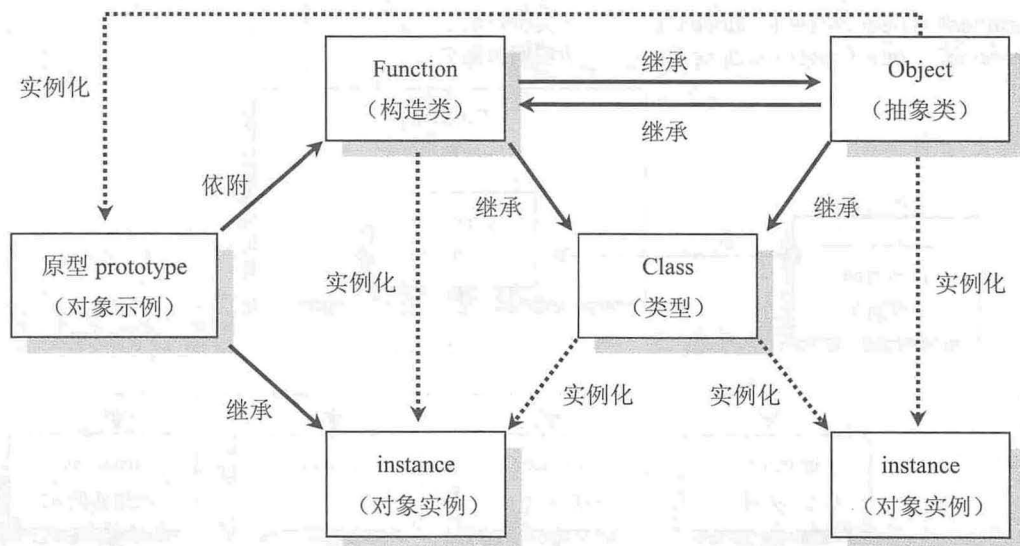


图 17.3 泛类、类型、原型和对象实例之间的关系

17.9.2 原型特性

先看一个示例：

```
function f(){                                //声明一个构造类型
    this.a = 1;                             //为构造类型声明一个本地属性
    this.b = function(){                   //为构造类型声明一个本地方法
        return this.a;
    };
}
```

```
var e = new f();           //实例化构造类型
alert(e.a);                //调用实例对象的属性 a，返回 1
alert(e.b());              //调用实例对象的方法 b，提示 1
```

构造函数 f() 定义了两个本地特性，分别是属性 a 和方法 b()。当构造函数实例化后，实例对象继承了构造函数的本地特性。此时还可以在本地修改实例对象的属性 a 和方法 b()。代码如下：

```
e.a = 2;  
alert(e.a);  
alert(e.b());
```

构造函数的本地属性和方法可以继承给实例对象:

function f(){	//声明一个构造类型
this.a = 1;	//为构造类型声明一个本地属性
}	
var e =new f();	//实例 e
var g =new f();	//实例 g
alert(e.a);	//返回值为 1，说明它继承了构造函数的初始值
alert(g.a);	//返回值为 1，说明它继承了构造函数的初始值
e.a = 2;	//修改实例 e 的属性 a 的值
alert(e.a);	//返回值为 2，说明实例 e 的属性 a 的值改变了
alert(g.a);	//返回值为 1，说明实例 g 的属性 a 的值没有受影响

上面的示例中，如果使用本地属性，则实例对象之间就不会相互影响。但是如果希望统一修改实例对

象中包含的本地属性值，就需要一个个修改了。this 关键字也可以说明问题，因为 this 是一个动态值，在不同实例对象中，this 分别指向当前对象，而不是构造函数的属性或原型对象。

正是在这种背景下，JavaScript 提出了原型的概念。这样修改任何一个实例中的原型属性值，则该构造函数的所有实例的原型值都会随之发生变化。例如，继续以上面的示例为基础进行说明：

```
function f(){           //声明一个构造类型
    f.prototype.a = 1;  //为构造类型声明一个原型属性
}
var e = new f();        //实例 e
var g = new f();        //实例 g
a alert(e.a);           //返回值为 1，说明它继承了构造函数的初始值
alert(g.a);             //返回值为 1，说明它继承了构造函数的初始值
f.prototype.a = 2;      //修改原型属性值
alert(e.a);             //返回值为 2，说明实例 e 的属性 a 的值改变了
alert(g.a);             //返回值为 2，说明实例 g 的属性 a 的值改变了
```

从上面的示例可以看到，原型属性值会影响所有实例对象的属性值，对于原型方法也相同，这里就不再说明了。对于原型属性或原型方法的声明比较灵活，都可以在构造函数结构体内。例如：

```
function f(){           //声明一个空的构造类型
    f.prototype.a = 1;   //在结构体外为构造类型声明一个原型属性
    f.prototype.b = function(){ //在结构体外为构造类型声明一个原型方法
        return f.prototype.a; //返回原型属性值
    }
}
```

prototype 属性是属于构造函数的，所以必须使用构造函数通过点语法来调用 prototype 属性，再通过 prototype 属性来访问原型对象内的数据。原型属性与本地特性之间的关系如图 17.4 所示。

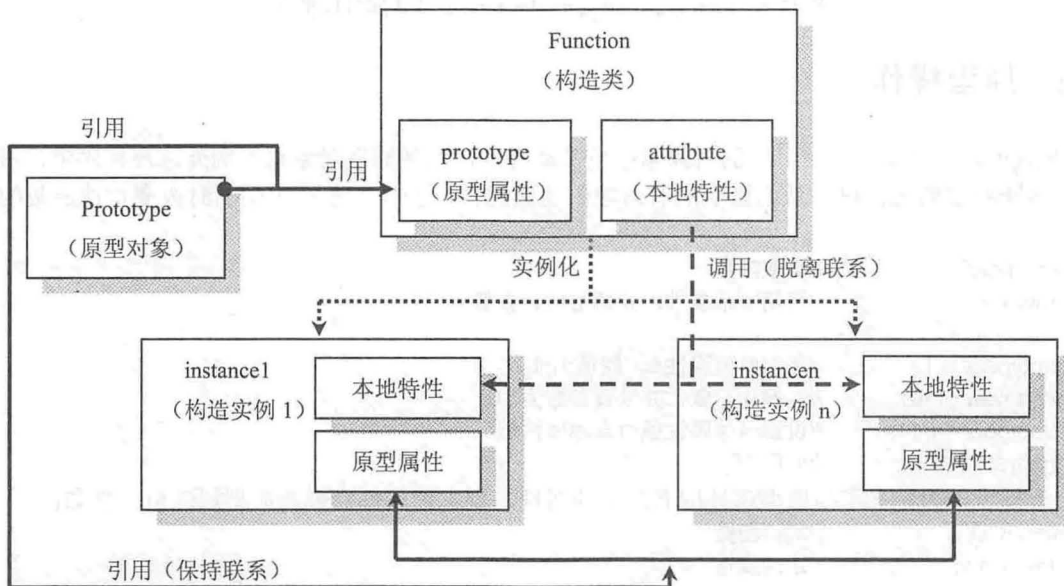


图 17.4 原型属性与本地特性之间的关系

Object 和 Function 都可以定义原型，此时 Object 被视为 Function 的子类。下面的示例说明 Object 和 Function 原型的异同，它们的属性与原型关系如图 17.5 所示。

```
Object.prototype.a = 1; //声明 Object 的原型属性 a 的值为 1
Function.prototype.a = 2; //声明 Function 的原型属性 a 的值为 2
alert(Object.a);         //返回 2，说明属性 a 指向 Function 构造函数的原型
alert(Function.a);       //返回 2，说明属性 a 指向 Function 构造函数的原型
```



```

var o = {}           //空的对象直接量
alert(o.a);          //返回 1, 说明属性 a 指向 Object 构造函数的原型
var f = Object;      //引用 Object 构造函数
alert(f.a);          //返回 2, 说明属性 a 指向 Function 构造函数的原型
var f1 = new Function(); //实例化 Function 对象
alert(f1.a);         //返回 2, 说明属性 a 指向 Function 构造函数的原型
var o1 = new Object(); //实例化 Object 对象
alert(o1.a);         //返回 1, 说明属性 a 指向 Object 构造函数的原型

```

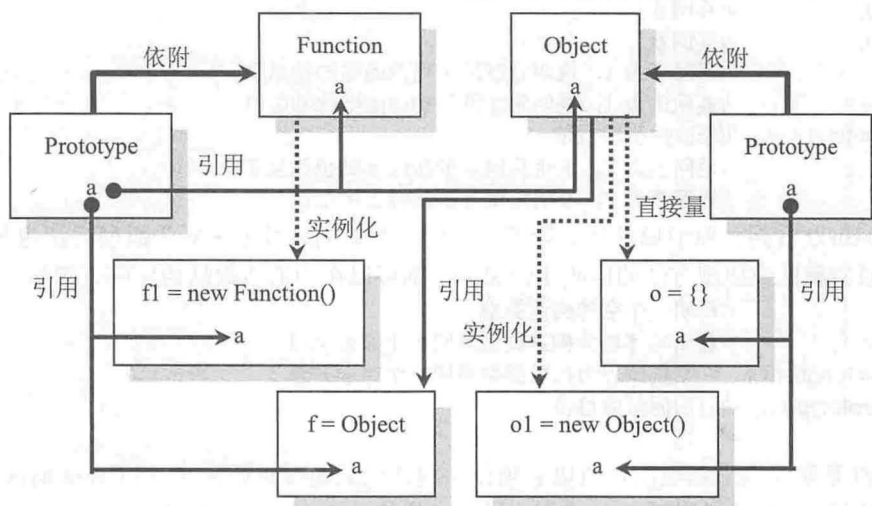


图 17.5 Function、Object、Prototype 及其属性间的关系

17.9.3 原型操作

JavaScript 是一种动态语言，它的对象系统也是动态的，当然原型对象也不例外。在程序中，可以根据需要改变原型的属性值，从而影响所有对象的实例属性。例如，在程序中随时改变构造函数的原型属性值：

```

function p(x){       //构造函数
    this.x = x;      //声明本地属性，并初始化为参数 x
}
p.prototype.x = 1    //添加原型属性 x，赋值为 1
var p1 = new p(10);  //实例化对象，并设置参数为 10
p.prototype.x = p1.x //设置原型属性值为本地属性值
alert(p.prototype.x); //返回 10

```

如果给构造函数定义了与原型属性同名的本地属性，则本地属性会覆盖原型属性值。例如：

```

function p(x){       //构造函数
    this.x = x;      //本地属性
}
p.prototype.x = 1    //原型属性
var p1 = new p(10);  //实例化对象
alert(p1.x);         //返回 10, 本地属性覆盖原型属性

```

但是原型属性并没有删除，它依然存在，仅是被同名本地属性覆盖了。如果使用 `delete` 运算符删除本地属性，则原型属性依然显示出来。例如，在上面的示例基础上删除本地属性，则会发现原型属性：

```

delete p1.x;         //删除本地属性
alert(p1.x);         //返回 1, 显示被覆盖的原型属性

```

利用它可以在对象原型与本地属性之间自由切换, 从而实现各种复杂功能的编程。

```
function p(x,y,z){           //构造函数
    this.x = x;              //声明本地属性 x 并赋参数 x 的值
    this.y = y;              //声明本地属性 y 并赋参数 y 的值
    this.z = z;              //声明本地属性 z 并赋参数 z 的值
}
p.prototype.del = function(){ //定义原型方法
    for(var i in this){      //遍历本地对象, 删除实例对象内的所有属性和方法
        delete this[i];
    }
}
p.prototype = new p(1,2,3);  //实例化构造函数 p, 并把实例对象传递给原型对象
var p1 = new p(10,20,30);    //实例化构造函数 p 为 p1
alert(p1.x);                  //返回 10, 本地属性 x 的值
alert(p1.y);                  //返回 20, 本地属性 y 的值
alert(p1.z);                  //返回 30, 本地属性 z 的值
p1.del();                     //调用原型方法, 删除所有本地属性
alert(p1.x);                  //返回 1, 原型属性 x 的值
alert(p1.y);                  //返回 2, 原型属性 y 的值
alert(p1.z);                  //返回 3, 原型属性 z 的值
```

上面的示例定义了构造函数 p, 并声明 3 个本地属性。然后实例化构造函数, 并把实例对象赋值给构造函数的原型对象。同时定义原型方法 del(), 该方法将删除实例对象的所有本地属性和方法。最后, 分别调用属性 x、y 和 z, 则返回的是本地属性值, 调用方法 del(), 删除所有本地属性, 则再次调用属性 x、y 和 z, 返回的是原型属性值。

1. 利用原型为对象设置默认值

当原型属性与本地属性相同时, 它们之间可以出现交流现象。利用这种现象为对象初始化默认值。

例如:

```
function p(x){               //构造函数
    if(x)                     //如果参数存在, 则使用该参数设置属性, 该条件是关键
        this.x = x;          //使用参数初始化本地属性 x 的值
}
p.prototype.x = 0;           //利用原型属性, 设置本地属性 x 的默认值
var p1 = new p();             //实例化一个没有带参数的对象
alert(p1.x);                  //返回 0, 即显示本地属性的默认值
var p2 = new p(1);            //再次实例化, 传递一个新的参数
alert(p2.x);                  //返回 1, 即显示本地属性的初始化值
```

2. 利用原型间接实现本地数据备份

把本地对象的数据完全赋值给原型对象, 相当于为该对象定义一个副本, 通俗地说就是备份对象。这样当对象属性被修改时, 可以通过原型对象来恢复本地对象的初始值。例如:

```
function p(x){               //构造函数
    this.x = x;
}
p.prototype.backup = function(){
    //原型方法, 备份本地对象的数据到原型对象中
    for(var i in this){
        p.prototype[i] = this[i];
    }
}
```

```

var p1 = new p(1);           //实例化对象
p1.backup();                 //备份实例对象中的数据
p1.x = 10;                   //改写本地对象的属性值
alert(p1.x)                  //返回 10, 说明属性值已经被改写
p1 = p.prototype;           //恢复备份
alert(p1.x)                  //返回 1, 说明对象的属性值已经被恢复到原始值

```

3. 利用原型设置只读属性

利用原型还可以为对象属性设置“只读”特性，这在一定程度上可以避免对象内部数据被任意修改的尴尬。下面这个示例演示了如何根据平面上两点坐标来计算它们之间的距离。构造函数 `p` 用来设置定位点坐标，当传递两个参数值时，会返回以参数为坐标值的点，如果省略参数则默认点为原点 (0,0)。而在构造函数 `l` 中通过传递的两点坐标对象，计算它们的距离。

如果无意间修改了构造函数 `l` 的方法 `b()` 或 `e()` 的值，则构造函数 `l` 中的 `length()` 方法的计算值也随之发生变化。这种动态效果对于需要动态跟踪两点坐标变化来说，是非常必要的。但是，这里并不需要当初初始化实例之后，随意地被改动坐标值。毕竟方法 `b()` 和 `f()` 与参数 `a` 和 `b` 是没有多大联系的。但是它们的值却同时指向同一个对象的引用指针。

```

function p(x,y){             //求坐标点构造函数
    if(x) this.x = x;        //初始 x 轴值
    if(y) this.y = y;        //初始 y 轴值
    p.prototype.x = 0;       //默认 x 轴值
    p.prototype.y = 0;       //默认 y 轴值
}
function l(a,b){             //求两点距离构造函数
    var a = a;                //参数私有化
    var b = b;                //参数私有化
    var w = function(){      //计算 x 轴距离, 返回对函数的引用
        return Math.abs(a.x - b.x);
    }
    var h = function(){      //计算 y 轴距离, 返回对函数的引用
        return Math.abs(a.y - b.y);
    }
    this.length = function(){ //计算两点距离, 计算中使用小括号调用私有方法 w()和 h()
        return Math.sqrt(w()*w() + h()*h());
    }
    this.b = function(){      //获取起点坐标对象
        return a;
    }
    this.e = function(){      //获取终点坐标对象
        return b;
    }
}
var p1 = new p(1,2);         //实例化 p 构造函数, 声明一个点
var p2 = new p(10,20);       //实例化 p 构造函数, 声明另一个点
var l1 = new l(p1,p2);       //实例化 l 构造函数, 传递两点对象
alert(l1.length())           //返回 20.12461179749811, 调用 length()方法计算两点距离
l1.b().x = 50;               //不经意改动方法 b()的一个属性为 50
alert(l1.length())           //返回 43.86342439892262, 说明改动影响到两点距离值

```

为了避免因为改动方法 `b()` 的属性 `x` 值会影响两点距离，可以在方法 `b()` 和 `e()` 中新建一个临时性的构造类，设置该类的原型为 `a`，然后实例化构造类并返回，这样就阻断了方法 `b()` 与私有变量 `a` 的直接联系，它们之间仅就是值的传递，而不是对对象 `a` 的引用，从而避免因为方法 `b()` 的属性值变化，而影响私有对象 `a`

的属性值。代码如下：

```

this.b = function(){           //方法 b()
    function temp(){};         //临时构造类
    temp.prototype = a;        //把私有对象传递给临时构造类的原型对象
    return new temp();         //返回实例化后的对象，阻断直接返回 a 所出现的引用关系
}
this.e = function(){           //方法 fe()
    function temp(){};         //临时构造类
    temp.prototype = a;        //把私有对象传递给临时构造类的原型对象
    return new temp();         //返回实例化后的对象，阻断直接返回 a 所出现的引用关系
}

```

还有一种方法是在给私有变量 *w* 和 *h* 赋值时，不是赋值函数，而是函数调用表达式。这样私有变量 *w* 和 *h* 存储的是值类型数据，而不是对函数结构的引用，从而就不再受后期相关属性值的影响。代码如下：

```

function l(a,b){               //求两点距离构造函数
    var a = a;                 //参数私有化
    var b = b;                 //参数私有化
    var w = function(){        //计算 x 轴距离，返回函数表达式的计算值
        return Math.abs(a.x - b.x);
    }()
    var h = function(){        //计算 y 轴距离，返回函数表达式的计算值
        return Math.abs(a.y - b.y);
    }()
    this.length = function(){   //计算两点距离，直接使用私有变量 w 和 h 来计算
        return Math.sqrt(w()*w() + h()*h());
    }
    this.b = function(){        //获取起点坐标对象
        return a;
    }
    this.e = function(){        //获取终点坐标对象
        return b;
    }
}

```

4. 利用原型进行批量复制

先看一个示例：

```

function f(x){                 //构造函数
    this.x = x;                //声明本地属性
}
var a = [];                    //声明数组
for( var i = 0; i < 100; i++ ){ //使用 for 循环结构批量复制构造类 f 的同一个实例
    a[i] = new f( 10 );        //把实例分别存入数组
}

```

上面的代码演示了如何复制 100 次同一个实例对象。这种做法本无可非议，但是如果在后期修改数组中每个实例对象时，就会非常麻烦。现在可以尝试使用原型来进行批量复制操作。代码如下：

```

function f(x){                 //构造函数
    this.x = x;                //声明本地属性
}
var a = [];                    //声明数组
function temp(){}              //定义一个临时的空构造类 temp
temp.prototype = new f( 10 );  //把构造类 f 实例化，并传递给构造类 temp 的原型对象
for( var i = 0; i < 100; i++ ){ //使用 for 循环结构批量复制临时构造类 temp 的同一个实例
}

```



```
a[i] = new temp();           //把实例分别存入数组
}
```

把构造类 `f` 的实例存储在临时构造类的原型对象中，然后通过临时构造类 `temp` 实例来传递复制的值。这样，要想修改数组的值，只需要修改类 `f` 的原型即可，从而避免逐一修改数组中的每个元素。

17.9.4 定义静态原型

使用 `Prototype` 定义的属性和方法被称为静态属性和静态方法，也称为原型属性和原型方法。而对于使用 `this` 定义的属性和方法被称为本地属性和方法，或称为公有属性和公有方法。

原型最有价值的应用就是为构造类定义静态方法。由于对象的方法比较固定，不轻易被修改，这是与对象属性最大的不同。同时使用原型的静态特性定义方法，能够在构造函数实例化过程中每次都需要重新赋值的麻烦，从而节省了系统资源和时间。例如：

```
function f(x,y){              //构造函数
    this.x = x;
    this.y = y
}
f.prototype.add = function(){  //原型方法
    return this.x + this.y;
}
```

使用本地方法进行定义：

```
function f(x,y){              //构造函数
    this.x = x;
    this.y = y;
    this.add = function(){    //本地方法
        return this.x + this.y;
    }
}
```

使用原型方法能够避免构造函数实例化过程中重复调用本地方法的弊端，这样就能够节省大量系统资源。尽管使用 `prototype` 与 `this` 关键字定义的属性和方法在形式上是一样的，但是它们的本质是不同的。使用 `prototype` 能够避免每次调用构造函数时对 `this.add()` 本地方法进行赋值操作和函数构造。如果程序中反复调用，就会影响执行效率和占用大量系统资源。

一般构造函数尽量采用原型方法，避免定义本地方法，除非该方法将要访问对象的私有成员或者引用了构造函数环境中的闭包。尽管使用 `prototype` 或 `this` 定义属性和方法时，在用法上没有差异，很多时候会把它们混淆在一起，但是 `prototype` 在数据存储上具有静态的特性，也就是说，所有实例对象都引用同一个原型指针。当然，原型最大的价值是它在 `JavaScript` 类继承中的广泛应用。

17.9.5 原型域和原型域链

实例对象在读取某个属性时，总是先检查自身域的属性。如果存在这样的属性，则会返回该属性值，否则就会往上检索 `prototype` 原型域，并返回 `prototype` 原型域中的同名属性。`prototype` 原型域允许原型属性引用任何类型的对象。因此，如果在 `prototype` 原型域中没有找到指定的属性，则 `JavaScript` 将会根据引用关系，继续向外查找 `prototype` 原型域所指向对象的 `prototype` 原型域，直到对象的 `prototype` 域为它自己，或者出现循环为止。例如，下面的这个示例演示了对象属性查找的 `prototype` 规律：

```
function a(x){                //构造函数 a()
    this.x = x;
}
```

```

a.prototype.x = 0;           //原型属性 x 的值为 0
function b(x){               //构造函数 b()
    this.x = x;
}
b.prototype = new a(1);      //原型对象为构造函数 a()的实例
function c(x){               //构造函数 c()
    this.x = x;
}
c.prototype = new b(2);      //原型对象为构造函数 b()的实例
var d = new c(3);            //实例化构造函数 c()
alert(d.x);                  //调用实例对象 d 的属性 x, 返回值为 3
delete d.x;                  //删除实例对象的本地属性 x
alert(d.x);                  //调用实例对象 d 的属性 x, 返回值为 2
delete c.prototype.x;        //删除 c 类的原型属性 x
alert(d.x);                  //调用实例对象 d 的属性 x, 返回值为 1
delete b.prototype.x;        //删除 b 类的原型属性 x
alert(d.x);                  //调用实例对象 d 的属性 x, 返回值为 0
delete a.prototype.x;        //删除 a 类的原型属性 x
alert(d.x);                  //调用实例对象 d 的属性 x, 返回值为 undefined

```

原型链表现了 JavaScript 面向对象的本质。每个对象实例都有属性成员用于指向它的构造类（父对象）的原型（Prototype），可以把这种层层指向父原型的关系称为原型链（Prototype Chian），如图 17.6 所示。

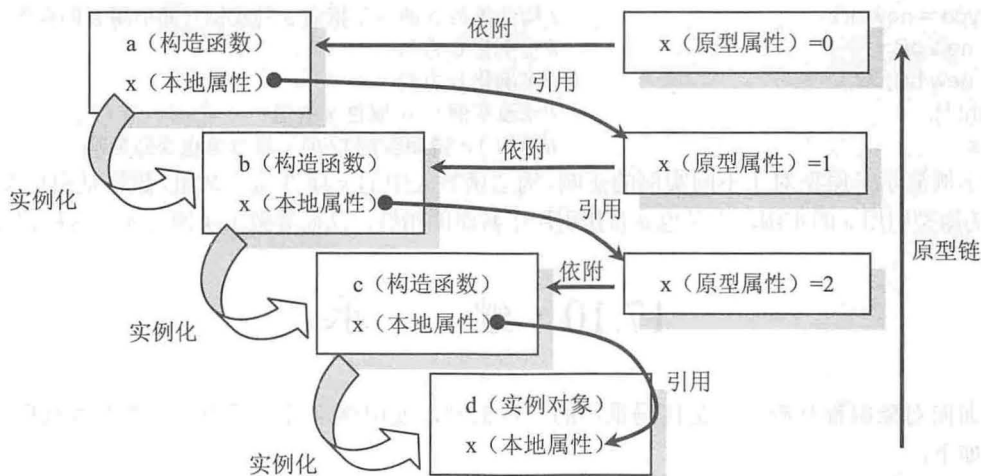


图 17.6 原型链检索示意图

在 JavaScript 中，一切都是对象，函数是第一型。Function 和 Object 都是函数的实例。构造函数的父原型指向 Function 的原型，Function.prototype 的父原型是 Object 的原型，Object 的父原型也指向 Function 的原型，Object.prototype 是所有父原型的顶层。

```

Function.prototype.a = function(){           //Function 原型方法
    alert( "Function" );
}
Object.prototype.a = function(){              //Object 原型方法
    alert( "Object" );
}
function f(){                                 //构造函数 f()
    this.a = "a";
}

```

```

f.prototype = {                                //构造函数 f()的原型方法
    w : function(){
        alert( "w" );
    }
}
alert( f instanceof Function );                //返回 true, 说明 f 是 Function 的实例
alert( f.prototype instanceof Object );        //返回 true, 说明 f 的原型也是对象
alert( Function instanceof Object );           //返回 true, 说明 Function 是 Object 的实例
alert( Function.prototype instanceof Object ); //返回 true, 说明 Function 的原型是 Object 的实例
alert( Object instanceof Function );           //返回 true, 说明 Object 是 Function 的实例
alert( Object.prototype instanceof Function ); //返回 false, 说明 Object.prototype 是所有父原型的顶层

```

prototype 是一种模拟面向对象的机制,它通过原型实现对类与实例之间的关系,并进行管理。以 prototype 机制模拟继承机制是一种原型继承,它也是 JavaScript 的核心功能之一。但是 prototype 的真正价值在于它能够以对象结构为载体,创建大量的实例,这些实例能够在构造类下实现共享。利用 prototype 的这个特性模拟对象的继承机制。原型域不是一种值复制行为,而是一种值引用现象。原型的引用关系也容易带来副作用。如果改变某个原型上的引用类型的属性值,将会影响到该原型作用的所有实例对象。

```

function a(){                                //构造函数 a
    this.x = [];                             //本地属性 x 值为数组
}
function b(){                                //构造函数 b
}
b.prototype = new a();                      //构造函数 b 的原型指向 a 的实例 (即引用 a 的实例)
var f1 = new b();                            //实例化 b 为 f1
var f2 = new b();                            //实例化 b 为 f2
f1.x.push(1);                                //修改实例 f1 中属性 x 的值
alert(f2.x);                                //返回 1, 说明实例 f2 的 x 属性值也受到影响

```

上面的示例演示了原型对于不同实例的影响。构造函数 a 中的 x 属性值为数组,数组是引用类型的数据,而构造函数的原型引用 a 的实例,于是也就直接引用了数组的指针,从而导致了实例之间的这种相互影响现象。

17.10 继 承

继承是面向对象编程基础,它是代码重用的一种机制,使用继承可以减少重复性开发代码。实现继承的主要作用如下:

- ☒ 子类实例可以共享超类属性和方法。
- ☒ 子类可以覆盖和扩展超类属性和方法。
- ☒ 子类和超类都是子类实例的类型。

对象系统的继承机制有 3 种实现方案:基于类、基于原型、基于元类。JavaScript 采用原型继承来实现对象系统,JavaScript 不支持类 (Class) 概念,使用构造器 (Constructor) 机制实现类的特性。原型继承是 JavaScript 最重要的语言特性之一。

JavaScript 实现继承的方法不止一种,主要包括类继承、原型继承、实例继承、复制继承和混合继承等,因为 JavaScript 继承机制不是规范的语法。

17.10.1 原型继承

原型继承是一种简化的继承机制,也是 JavaScript 最通用的继承方式。在原型继承中,类和实例概念被淡化,不用类实例化对象,而是通过直接定义对象,并被其他对象引用,这样就形成了一种继承关系,其

中引用对象被称为原型对象 (Prototype Object)。JavaScript 能够根据原型链实现对象之间的这种继承关系。例如：

```
function A(x){           //A 类
    this.x1= x;           //A 的本地属性 x1
    this.get1 = function(){ //A 的本地方法 get1()
        return this.x1;
    }
}
function B(x){           //B 类
    this.x2 = x;           //B 的本地属性 x2
    this.get2 = function(){ //B 的本地方法 get2()
        return this.x2 + this.x2;
    };
}
B.prototype = new A(1);   //原型对象继承 A 的实例
function C(x){           //C 类
    this.x3 = x;           //C 的本地属性 x3
    this.get3 = function(){ //C 的本地方法 get3()
        return this.x2 * this.x2;
    };
}
C.prototype = new B(2);   //原型对象继承 B 的实例
```

在上面示例中，分别定义了 3 个函数，然后通过原型继承方法把它们串连在一起，这样 C 能够继承 B 和 A 函数的成员，而 B 能够继承 A 的成员。prototype 的最大特点就是能够允许对象实例共享原型对象的成员。因此如果把某个对象作为一个类型的原型，那么这个类型的实例以这个对象为原型。这个时候，实际上这个对象的类型也可以作为那些以这个对象为原型的实例的类型。此时，可以在 C 的实例中调用 B 和 A 的成员。例如：

```
var b = new B(2);         //实例化 B
var c = new C(3);         //实例化 C
alert(b.x1);              //在实例对象 b 中调用 A 的属性 x1，返回 1
alert(c.x1);              //在实例对象 c 中调用 A 的属性 x1，返回 1
alert(c.get3());          //在实例对象 c 中调用 C 的方法 get3()，返回 9
alert(c.get2());          //在实例对象 c 中调用 B 的方法 get2()，返回 4
```

基于原型的编程是面向对象编程的一种特定形式。在这种编程模型中，不需要声明静态类，而是通过复制已经存在的原型对象来实现继承关系。因此，基于原型的模型没有类的概念，原型继承中的类仅是一种模拟，或者说是沿用面向对象编程的概念。

原型继承显得非常简单，其优点也是结构简练，不需要每次构造都调用父类的构造函数，且不需要通过复制属性的方式就能快速实现继承。但是它也存在以下几个缺点：

- ☑ 每个类型只有一个原型，所以它不直接支持多重继承。
- ☑ 它不能很好地支持多参数或者动态参数的父类。也许在原型继承阶段，用户还不能决定以什么参数来实例化构造函数。
- ☑ 使用不够灵活。用户需要在原型声明阶段实例化父类对象，并把它作为当前类型的原型，这限制了父类实例化的灵活性，很多时候无法确定父类对象实例化的时机和场所。
- ☑ prototype 属性固有的副作用。

17.10.2 类继承（上）

类继承也称为构造函数继承，或称对象模拟法。其表现形式是：在子类中执行父类的构造函数。其实

现本质是：构造函数也是函数，与普通函数相比，它只不过是一种特殊结构的函数而已。所以可以为一个构造函数（如 A）的方法赋值为另一个构造函数（如 B），然后调用该方法，使构造函数 A() 在构造函数 B() 内部被执行，这时构造函数 B() 就拥有构造函数 A() 中定义的属性和方法，这就是所谓 B 类继承 A 类。例如：

```
function A(x){           //构造函数 A()
    this.x = x;           //本地属性 x
    this.say = function(){ //本地方法 say()
        alert(this.x);
    }
}

function B(x,y){         //构造函数 B()
    this.m = A;           //把构造函数 A 作为一个普通函数引用给临时方法 m()
    this.m(x);            //把当前构造函数参数 x 作为值传递给构造函数 A，并执行
    delete this.m;        //清除临时方法
    this.y = y;           //本地属性 y
    this.call = function(){ //本地方法 call()
        alert(this.y);
    }
}

var a = new A(1);         //实例化 A
var b = new B(2,3);       //实例化 B
a.say();                  //调用实例化 A 的方法 say(), 返回 1
b.say();                  //在 B 类中调用 A 类的方法 say(), 返回 2，说明继承成功
b.call();                 //用实例化 B 的方法 call(), 返回 3
```

构造函数能够使用 this 关键字为所有属性和方法赋值。在默认情况下，关键字 this 引用的是构造函数当前创建的对象。不过在这个方法中，this 不是指向当前正在使用的实例对象，而是调用构造函数的方法所属的对象，即构造函数 B()。此时，构造函数 A() 已经不是构造函数结构了，而被视为一个普通可以执行函数。

这种函数之间的引用和执行，正是类继承的基础，可以设计为一个类继承多个类，只要遵循构造函数仅是一个函数，this 关键字是一个动态指针即可。例如，下面类 C 继承了类 A 和类 B 的所有成员：

```
function A(x){           //构造函数 A()
    this.x = function(){ //本地方法 x()
        alert(x);
    }
}

function B(y){           //构造函数 B()
    this.y = function(){ //本地方法 y()
        alert(y);
    }
}

function C(x,y){         //构造函数 C()
    this.m = A;           //引用构造函数 A()
    this.m(x);            //传递参数 x，并进行调用
    this.m = B;           //引用构造函数 B()
    this.m(y);            //传递参数 y，并进行调用
}

var c = new C(2,3);      //实例化类 C
c.x();                   //调用继承类 A 的方法 x(), 返回 2
c.y();                   //调用继承类 B 的方法 y(), 返回 3
```

在上面的示例中，没有使用 delete 运算符删除临时方法 m()，而是通过覆盖的方式代替，不会影响继承关系。

在复杂编程中，是不会使用上面方法来定义类继承的。因为这种设计模式太随意，缺乏严密性。严谨

的设计模式应该考虑到各种可能存在的情况和类继承关系中的相互耦合性。例如：

```
function A(x){           //构造函数 A()
    this.x = x;          //构造函数 A()的本地属性 x
}
A.prototype.getx = function(){ //构造函数 A()的原型方法 getx ()
    return this.x;
}
```

在上面的代码中，先创建一个构造函数，它相当于一个类，类名是构造函数的名称 A。在结构体内使用 `this` 关键字创建本地属性 `x`。方法 `getx()` 被放在类的原型对象中，它也就成为公共方法。然后，借助 `new` 运算符调用构造函数，返回的结果是新创建的对象实例：

```
var a1 = new A(1);       //实例化类 A 为对象 a1
最后，对象 a1 就可以继承类 A 的本地属性 x，还可以访问类 A 的原型方法 getx():
alert(a1.x);             //继承类 A 的属性 x
alert(a1.getx());        //引用类 A 的方法 getx()
```

上面的代码是一个简单的类演示例子。下面再创建一个类 B，继承类 A：

```
function B(x,y){         //构造函数 B()
    this.y = y;           //构造函数 B()的本地属性
    A.call(this,x);       //在构造函数 B()中调用超类 A，实现绑定
}
B.prototype = new A();   //设置原型链，建立继承关系
B.prototype.constructor = B; //恢复 B 的原型对象的构造函数为 B
B.prototype.gety = function(){ //构造函数 B()的原型方法
    return this.y;
}
```

在 JavaScript 中实现类的继承，需要考虑和设置以下 3 点：

- ☑ 在构造函数 B() 的结构体内，使用函数 `call()` 调用构造函数 A()，把 B 的参数 `x` 传递给调用函数。让 B 能够继承 A 的所有属性和方法，即 `A.call(this,x)`。
- ☑ 在构造函数 A() 和 B() 之间建立原型链，即 `B.prototype = new A()`。有关原型链的技术问题，曾经在上面的小节中讲解过。在 JavaScript 中每个构造类都有一个名为 `prototype` 的属性，该属性指向一个对象。当在访问对象某个成员时，如果在当前域中没有找到，则会根据 `prototype` 属性指向的对象并沿着这个原型链不断向上查找，直到找到为止，否则只检索到顶级域链。因此，为了实现类之间的继承，必须保证它们是原型链上的上下级关系，即设置子类的 `prototype` 属性指向超类的一个实例即可。
- ☑ 恢复 B 的原型对象的构造函数，即 `B.prototype.constructor = B`。当定义构造函数时，其原型对象（`prototype` 属性值）默认是一个 `Object` 类型的一个实例，其构造器（`constructor` 属性值）会被默认设置为该构造函数本身。如果改动 `prototype` 属性值，使其指向另一个对象，那么新对象就不会拥有原来的 `constructor` 属性值，所以必须重新设置 `constructor` 属性值。

此时，就可以在子类 B 的实例对象中调用超类 A 的属性和方法：

```
var f2 = new B(10,20); //实例化子类 B
alert(f2.getx());      //在实例中调用超类的方法 getx(), 返回 10
alert(f2.gety());      //在实例中调用子类自己的方法 gety(), 返回 20
```

下面是一个更复杂的多重继承的实例：

```
//基类 A
function A(x){           //构造函数 A()
    this.get1 = function(){ //本地方法，获取参数值
        return x;
    }
}
```

```

}
A.prototype.has = function(){           //原型方法，判断 getI()方法返回值是否为 0 (false)
    return !( this.getI() == 0 );
}
//超类 B
function B(){                             //构造函数 B()
    var a = [];                           //私有数组 a
    a = Array.apply( a, arguments );     //把参数数组传入数组 a 中
    A.call( this, a.length );            //在当前对象中调用 A 类，并把参数数组长度传递给它
    this.add = function(){               //本地方法，把参数数组补加到数组 a 中，并返回
        return a.push.apply( a, arguments );
    }
    this.geta = function(){              //本地方法，返回数组 a
        return a;
    }
}
B.prototype = new A();                   //设置 B 类的原型为 A 类的实例，从而建立原型链
B.prototype.constructor = B;            //恢复 B 类原型对象的构造器
B.prototype.str = function(){           //原型方法，把数组转换为字符串并返回
    return this.geta().toString();
}
//子类 C
function C(){                             //构造函数
    B.apply( this, arguments );          //在当前对象中调用 B 类，并把参数数组长度传递给它
    this.sort = function(){              //本地方法，以字符顺序对数组进行排序
        var a = this.geta();            //获取数组的值
        a.sort.apply( a, arguments );   //调用数组排序方法 sort()对数组进行排序
    }
}
C.prototype = new B();                   //设置 C 类的原型为 B 类的实例，从而建立原型链
C.prototype.constructor = C;            //恢复 C 类原型对象的构造器
//超类 B 的实例继承类 A 的成员
var b = new B(1, 2, 3, 4);               //实例化 B 类
alert(b.getI());                         //返回 4，调用 A 类的方法 getI()
alert(b.has());                          //返回 true，调用 A 类的方法 has()
//子类 C 的实例继承类 B 和类 A 的成员
var c = new C(30, 10, 20, 40);           //实例化 C 类
c.add(6, 5);                             //调用 B 类方法 add()，补加数组
alert(c.geta());                         //返回数组 30,10 ,20,40 ,6,5
c.sort();                               //排序数组
alert(c.geta());                         //返回数组 10,20 ,30,40 ,5,6
alert(c.getI());                        //返回 4，调用 A 类的方法 getI()
alert(c.has());                         //返回 true，调用 A 类的方法 has()
alert(c.str());                         //返回 10,20 ,30,40 ,5,6

```

设计类 C 继承类 B，而类 B 又继承了类 A。A、B、C 3 个类之间的继承关系是通过在子类中调用父类的构造函数来维护的。如 C 类中添加语句行“B.apply(this, arguments);”，该行语句能够在 B 类中调用 A 类，并把 B 的参数传递给 A，从而使 B 类拥有 A 类的所有成员。同理，在 B 类中添加语句行“A.call(this, a.length);”，该行语句把 B 类的参数长度作为值传递给 A 类，并进行调用，从而实现 B 类拥有 A 类的所有成员。

从继承关系上看，B 类继承了 A 类的本地方法 getI()。为了确保它还能够继承 A 类的原型方法，还需要为它们建立原型链，从而实现原型对象的继承关系，方法是添加语句行“B.prototype = new A();”。同理，在

C 类中添加语句行 “C.prototype = new B();”，这样就可以把 A、B 和 C 通过原型链串连在一起，从而实现子类能够继承超类成员，甚至还可以继承基类的成员。这里的成员主要指类的原型对象包含的成员，当然它们之间也可以通过相互调用，实现对本地成员的继承关系。在为 B 类的原型指定为 A 类的实例前，不能再为其定义任何原型属性或方法，否则就会被覆盖。如果要扩展原型方法，只有在原型绑定之后，再定义扩展方法。

17.10.3 类继承（下）

在类继承实现过程中，call()和 apply()方法被频繁使用。call()和 apply()方法在用法和功能上都是相同的，唯一区别是第 2 个参数类型不同。第 1 个参数设置 this 所指向的对象，其他参数则被直接传递给调用函数。例如：

```
function f(){                                //普通函数 f()
    alert(this.x);
}
var o = {};                                  //对象直接量
o.x = true;                                  //定义对象属性 x
f.call(o);                                   //在对象 o 中调用函数 f()，提示信息为 true
```

上面的代码中，函数 f()被 o 调用，此时函数 f()中的 this 关键字指针就自动指向对象 o，所以当调用函数 f()时，this.x 就表示 o.x，即 x 是对象 o 的一个属性，从而提示信息为 true。

下面示例演示 call()不仅仅是调用函数，还是引用函数，相当于把一个函数传递给一个对象，从而使对象也拥有函数的功能，从而实现继承功能。例如，在上面示例的基础上，调整函数 f()和对象 o 的结构。

```
function f(){                                //构造函数 f()
    this.m = function(){                    //把提示信息封装在 f 类的方法 m()中
        alert(this.x);
    };
}
function o(){                                //构造函数 o()
    this.x = true;                          //本地属性 x
    f.call(this);                          //在当前对象中调用函数 f()
}
var o1 = new o();                          //实例化构造函数 f()
o1.m();                                     //调用实例的方法 m，提示信息为 true
```

通过 call()方法调用，函数 f()就成为函数 o()的一部分，o 拥有 f 的方法 m。call()方法具有创建继承关系的功能。在 this 关键字的辅助下，它很巧妙地把一个类的所有成员都传递给另一个类，使它们保持一种继承关系。

与 call()方法不同，apply()方法要求第 2 个参数为数组或者 arguments 参数集合，而 call()方法的第 2 个参数及后面可以是任意多个参数。总之，不管是数组参数，还是任意多个参数，它们最终都会被传递给调用函数。例如，如果把上面示例稍稍调整，就可以实现一个自定义数组排序功能：

```
function f(){                                //构造函数 f()
    this.m = function(){                    //把提示信息封装在 f 类的方法 m()中
        return x.sort();                  //返回排序后的参数，注意参数必须为数组类型
    };
}
function o(x){                                //构造函数 o()
    f.call(this,x);                       //在当前对象中调用函数 f()
```



```

}
var a = [2,3,5,1,6,8,7,4]           //声明数组直接量
var o1 = new o(a);                   //实例化类 o, 并把参数 a 传递进去
alert(o1.m());                       //返回提示信息 1,2,3, 4,5, 6,7,8
call()和 apply()方法的用法可以转换为另一种形式。例如:

```

```
f.call(this,x);
```

等价于:

```

this.tempMethod = f;                 //定义一个临时方法引用函数 f()
this.tempMethod(x);                  //传递值并调用该函数
delete this.tempMethod;              //删除临时方法

```

在面向对象编程中都有一套严格的封装机制,但 JavaScript 语言没有提供封装机制,只能通过间接的方法实现部分功能封装。下面就来尝试把类继承这样一个简单的功能封装起来,不过这里封装的主要是类继承的原型成员,而对于本地成员还需要采用其他方法实现继承。

首先,定义一个封装函数。设计入口为子类和超类对象,函数功能是子类能够继承超类的所有原型成员,不设计出口。代码如下:

```

function extend(Sub,Sup){             //类继承封装函数
    //其中参数 Sub 表示子类, Sup 表示超类
}

```

在函数体内,首先定义一个空函数 F(),用来实现功能中转。设计它的原型为超类的原型,然后把空函数的实例传递给子类的原型,这样就避免了直接实例化超类可能带来的系统负荷。因为在实际开发中,超类的规模可能会很大,如果实例化,会占用大量内存。

最后,恢复子类原型的构造器子类自己。同时,检测超类原型构造器是否与 Object 的原型构造器发生耦合。如果是,则恢复它的构造器为超类自身。代码如下:

```

function extend(Sub,Sup){             //类继承封装函数
    var F = function({});             //定义一个空函数
    F.prototype = Sup.prototype;       //设置空函数的原型为超类的原型
    Sub.prototype = new F();           //实例化空函数,并把超类原型引用传递给子类
    Sub.prototype.constructor = Sub;   //恢复子类原型的构造器为子类自身
    Sub.sup = Sup.prototype;           //在子类中定义一个本地属性存储超类原型,这样可以避免子类和超类耦合
    if(Sup.prototype.constructor == Object.prototype.constructor){ //检测超类原型构造器是否为自身
        Sup.prototype.constructor = Sup //类继承封装函数
    }
}

```

一个简单的功能封装函数就这样实现了。下面定义两个类,尝试把它们绑定为继承关系。

```

function A(x){                        //构造函数 A()
    this.x = x;                       //本地属性 x
    this.get = function(){            //本地方法 get()
        return this.x;
    }
}
A.prototype.add = function(){         //原型方法 add()
    return this.x + this.x;
}
A.prototype.mul = function(){         //原型方法 mul()
    return this.x * this.x;
}

```

```

}
function B(x){                                //构造函数 B()
    A.call(this,x);                          //在函数体内调用构造函数 A(), 实现内部数据绑定
}
extend(B,A);                                //调用类继承封装函数, 把 A 和 B 的原型捆绑在一起
var f = new B(5);                            //实例化类 B
alert(f.get())                              //继承类 A 的方法 get(), 返回 5
alert(f.add())                              //继承类 A 的方法 add(), 返回 10
alert(f.mul())                              //继承类 A 的方法 mul(), 返回 25

```

在继承类封装函数中, 应该使用 `Sub.sup = Sup.prototype`。例如:

```

extend(B,A);
B.prototype.add = function(){               //为 B 类定义一个原型方法
    return this.x + "" + this.x
}

```

上面代码是在调用封装函数之后, 再为 B 类定义一个原型方法, 该方法名与 A 类中原型方法 `add` 同名, 但是功能不同。如果此时测试程序, 子类 B 定义的原型方法 `add()` 将会覆盖超类 A 的原型方法 `add()`:

```

alert(f.add())                             //返回字符串 55, 而不是数值 10

```

在 B 类的原型方法 `add()` 中调用超类的原型方法 `add()`, 从而避免代码发生耦合。代码如下:

```

B.prototype.add = function(){               //定义子类 B 的原型方法 add()
    return B.sup.add.call(this);           //在函数内部调用超类方法 add()
}

```

17.10.4 实例继承

实例化类可以创建新的实例对象, 这个实例对象将继承类的所有特性。实例继承正是对于这种实例化过程的概括。类继承和原型继承在客户端中是无法继承 DOM 对象的, 同时它们也不支持继承系统静态对象、静态方法等。例如, 使用类继承法继承 `Date` 对象:

```

function D(){                                //自定义构造函数
    Date.apply(this,arguments);              //调用 Date 对象, 对其进行引用, 实现继承的目的
}
var d = new D();                             //实例化自定义构造函数
alert(d.toLocaleString());                  //返回[object Object]

```

上面示例演示说明, 使用类继承是无法实现对静态对象的继承的, 这是因为系统对象的结构比较特殊, 它不是简单的函数体结构, 声明、赋值和初始化等操作都进行了独立的封装, 所以也就无法实现在自定义构造函数中的那种操作。

使用原型继承法继承 `Date` 对象:

```

function D(){                                //自定义空构造函数
}
D.prototype = new Date();                   //把 Date 对象的实例赋值给 D 的原型对象
var d = new D();                           //实例化 D
alert(d.toLocaleString());                  //返回错误提示

```

上面示例演示说明了使用原型继承也无法实现相同的目的。

不过, 使用实例继承法能够实现对所有 JavaScript 核心对象的继承。例如, 在下面的示例中, 把 `Date` 对象的实例化过程和方法调用封装在一个函数中, 然后返回实例对象, 这样就可以解决核心静态对象无法继承的问题。

```

function D(){                                //封装函数

```

```

var d = new Date();           //实例化 Date 对象
d.get = function(){          //定义本地方法，间接调用 Date 对象的 toLocaleString()方法
    alert(d.toLocaleString());
}
return d;                     //返回实例对象
}
var d = new D();              //实例化封装函数
d.get();                      //调用本地方法，返回当前本地的日期和时间

```

构造函数是一种特殊结构的函数，它没有返回值，通过 `this` 关键字来初始化实例对象，并且会返回值。当然，在构造函数中可以增加 `return` 语句，为其设置一个返回值，这时返回值就是 `new` 运算符执行表达式的值。因此，通过在构造函数中完成对类的实例化操作，然后返回实例对象，这就是实例继承的由来。使用实例继承法能够实现对所有对象的继承，包括自定义类、核心对象和 DOM 对象等。

- ❑ 实例继承法无法传递动态参数。类的实例化操作是在封闭的函数体内实现的，而不能通过 `call()` 或 `apply()` 方法来传递动态参数。如果继承需要传递动态参数，则这种继承就会带来很多不便。
- ❑ 实例继承只能返回一个对象，与原型继承一样，不支持多重继承。
- ❑ 由于通过封装的方法把对象实例化，以及初始化操作都被封装在一个函数体内，最后通过对封装函数执行实例化操作来获取继承的对象。但是这种做法无法真正实现继承对象是封装类的实例，它仍然保持与原对象的实例关系。例如：

```

alert(d instanceof Date);    //返回 true，说明对象 d 是对象 Date 的实例
alert(d instanceof D);      //返回 false，说明对象 d 不是对象 D 的实例

```

17.10.5 复制继承

复制继承的设计思路是：利用 `for/in` 语句遍历对象成员，逐一复制给另一个对象，通过这种方式从而实现继承关系。例如，在下面的示例中，定义一个 `F` 类，其包含 4 个成员。然后实例化并把它的所有属性和方法都复制给一个空对象 `o`，这样对象 `o` 就拥有了 `F` 类的所有属性和方法。

```

function F(x,y){              //构造函数 F()
    this.x = x;                //本地属性 x
    this.y = y;                //本地属性 y
    this.add = function(){     //本地方法 add()
        return this.x + this.y;
    }
}
F.prototype.mul = function(){ //原型方法 mul()
    return this.x * this.y;
}
var f = new F(2,3)            //实例化构造函数，并进行初始化
var o = {}                    //定义一个空对象 o
for(var i in f){               //遍历构造函数的实例，把它的所有成员赋值给对象 o
    o[i] = f[i];
}
alert(o.x);                    //返回 2
alert(o.y);                    //返回 3
alert(o.add());                //返回 5
alert(o.mul());                //返回 6

```

可以封装复制继承法，使其具有较大的灵活性。代码如下：

```

Function.prototype.extend = function(o){           //为 Function 扩展复制继承的方法
    for(var i in o){                               //遍历参数对象
        this.constructor.prototype[i] = o[i];     //把参数对象成员复制给当前对象的构造函数原型对象
    }
}

```

上面的封装函数通过原型对象为 Function 核心对象扩展一个方法，该方法能够把指定的参数对象完全复制给当前对象的构造函数的原型对象。

this 关键字指向的是当前实例对象，而不是构造函数本身，所以要为其扩展原型成员，就必须使用 constructor 属性来指向它的构造器，然后通过 prototype 属性指向构造函数的原型对象。

然后，新建一个空的构造函数，并为其调用 extend() 方法把传递进来的 F 类的实例对象完全复制为原型对象成员。请注意，此时就不能够定义对象直接量，因为 extend() 方法只能够为构造函数结构复制继承。

```

var o = function();           //新建空白构造函数
o.extend(new F(2,3));         //调用复制继承方法

```

复制继承法不是真正的继承，它实际上是通过反射机制复制类对象的所有可枚举属性和方法来模拟继承。这种方法能够实现模拟多继承。不过，它的缺点也很明显：

- ☑ 由于是反射机制，复制继承法不能继承非枚举类型的方法。对于系统核心对象的只读方法和属性也是无法继承的。
- ☑ 通过反射机制来复制对象成员的执行效率会非常差。对象结构越庞大，这种低效就越明显。
- ☑ 当前类型如果包含同名成员，这些成员可能会被父类的动态复制所覆盖。
- ☑ 在多重继承的情况下，复制继承不能够清晰地描述出父类与子类的相关性。
- ☑ 只有当类实例化之后，才能够实现遍历成员和复制成员，所以它不能够灵活支持动态参数。
- ☑ 由于复制继承法仅是简单的引用赋值，如果父类的成员值包含引用类型，那么继承之后，与原型继承法一样容易带来很多副作用。

17.10.6 克隆继承

通过对象克隆方式来实现继承，这样就可以避免一个个复制对象成员所带来的低效率。具体方法如下。

首先，为 Function 对象扩展一个方法，该方法能够把参数对象赋值给一个空构造函数的原型对象，然后实例化构造函数，并返回实例对象，这样该对象就拥有构造函数包含的所有成员。代码如下：

```

Function.prototype.clone = function(o){           //对象克隆方法
    function Temp(){                               //新建空构造函数
        Temp.prototype = o;                       //把参数对象赋值给该构造函数的原型对象
    }
    return new Temp();                             //返回实例化后的对象
}

```

调用该方法来克隆对象。克隆方法返回的是一个空对象，不过它存储了指向给定对象的原型对象指针。这样就可以利用原型链来访问它们，从而在不同对象之间实现继承关系。代码如下：

```

var o = Function.clone(new F(2,3));               //调用 Function 对象的克隆方法
alert(o.x);                                       //返回 2
alert(o.y);                                       //返回 3
alert(o.add());                                   //返回 5
alert(o.mul());                                   //返回 6

```

17.10.7 混合继承

混合继承是把多种继承方法结合在一起使用，从而发挥各自优势，扬长避短，以实现各种复杂的应用。

其中最常见的形式是把类继承与原型继承混合使用，来解决类继承中存在的问题。

类继承与原型继承是两种截然不同的继承模式，它们生成对象的行为方式也是不同的。面向对象的开发人员对于类继承比较熟悉，几乎所有使用面向对象的 JavaScript 应用都用到了这种继承模式。但是，因为 JavaScript 中的类继承仅仅是对真正基于类继承的一种模仿，所以深入理解 JavaScript 的开发人员应该懂得原型继承的工作机制。

原型继承更能节约内存。原型链读取成员的方式使得所有克隆出来的对象都共享一个实例，只有在直接设置了某个克隆出来的对象的属性和方法时，情况才会有所变化。而在类继承方式中，创建的每一个对象在内存中都有自己的一套属性和方法副本。原型继承在这方面的节约效果很突出。这种继承也比类继承显得更为简练，读者不要把原型继承的简洁看作是简陋，它的力量蕴涵在其简洁性之中。

该使用类继承还是原型继承也许并不重要，更多时候是根据个人的使用习惯。有些人天生就容易被原型继承的简约身材所吸引，而另一些人却对更严谨的类继承情有独钟。不过，也可以把它们混合在一起使用。例如：

```
function A(x,y){           //构造函数 A()
    this.x = x;
    this.y = y;
}
A.prototype.add = function(){ //原型方法
    return this.x + this.y;
}
function B(x,y){           //构造函数 B()
    A.call(this,x,y);       //类继承实现
}
B.prototype = new A();     //原型继承实现
var b = new B(10,20);      //实例化 B
alert(b.x);                //返回 10
alert(b.y);                //返回 20
alert(b.add());             //返回 30
```

上面的示例把原型继承和类继承混用在一起，从而实现了一种比较完善的继承机制。也可以把原型继承与复制继承混合使用，也能够实现相同的继承效果。下面简单比较一下几种继承方法的异同，如表 17.8 所示。

表 17.8 继承方法综合比较

项 目	类 继 承	原 型 继 承	实 例 继 承	复 制 继 承	克 隆 继 承
原型属性	不继承	继承	继承	继承	继承
本地成员	继承	不继承	继承	继承	继承
多重继承	支持	不支持	不支持	支持	不支持
多参数	支持	不支持	不支持	不支持	不支持
执行效率	高	高	中	低	低
instanceof	false	true	false	false	false

17.10.8 多重继承

继承一般包括单向继承和多重继承两种模式。其中单向继承模式比较简单，每个子类有且仅有一个超类，而多重继承是一种比较复杂的继承模式。在这种模式中，一个子类可以拥有任意多个超类，如图 17.7 所示。其中 SupClass 表示超类，而 SubClass 表示子类。根据 UML 规定，类方框中间一栏为属性，下面一栏为方法，属性和方法后面的关键字表示成员的数据类型。

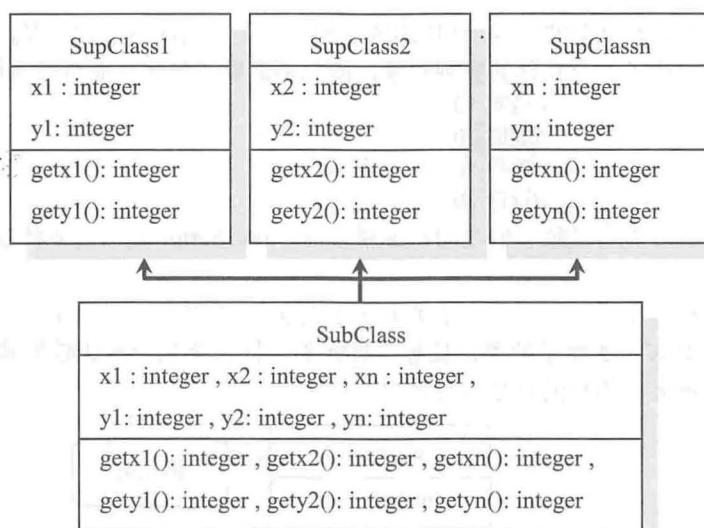


图 17.7 类的多重继承示意图

JavaScript 原型继承机制不支持多重继承，不过可以通过混合模式来实现多重继承。例如，设计 3 个类型 A、B、C，现在希望 C 能够同时继承 A 和 B 的属性和方法：

```
function A(x){                                //构造函数 A()
    this.x = x;                               //构造函数 A()的本地属性 x
}
A.prototype.getx = function(){                //构造函数 A()的原型方法 getx()
    return this.x;
}
function B(y){                                //构造函数 B()
    this.y = y;                               //构造函数 B()的本地属性 y
}
B.prototype.gety = function(){                //构造函数 B()的原型方法 gety()
    return this.y;
}
function C(x,y){                              //构造函数 C()
}
```

为了实现多重继承，唯一可以采用的方法是使用复制继承，或者结合类继承来实现。首先，定义一个复制继承扩展函数：

```
Function.prototype.extend = function(o){      //为 Function 扩展复制继承的方法
    for(var i in o){                           //遍历参数对象
        this.constructor.prototype[i] = o[i]; //把参数对象成员复制给当前对象的构造函数原型对象
    }
}
```

然后，使用类继承方法在 C 中调用 A 和 B：

```
function C(x,y){                              //类继承
    A.call(this,x);
    B.call(this,y);
}
```

再调用复制继承扩展函数来复制 A 和 B 的属性和方法：

```
C.extend(new A(10));                          //复制继承
C.extend(new B(20));                          //复制继承
```

此时，如果在 C 对象中，就可以调用 A 和 B 的属性和方法了。请注意，通过复制继承之后，C 不再是一个构造函数，它实际上变成了一个具体的实例对象，因此不需要实例化，而直接调用。

```
alert(C.x);           //返回 10
alert(C.y);           //返回 20
alert(C.getx());      //返回 10
alert(C.gety());      //返回 20
```

让一个类继承另一个类可能会导致它们之间产生耦合性，JavaScript 提供了多种技术途径来避免这种问题的发生，如掺元类等。

上面介绍了一个类继承多个类，如何让一个类被多个类继承，如图 17.8 所示。这种继承关系被形象地称为多亲继承，其中能够被多个类继承的类，被称为掺元类。掺元类是一种比较特殊的类形式，它一般不会被实例化或调用。定义掺元类的目的只是向其他类提供通用方法。

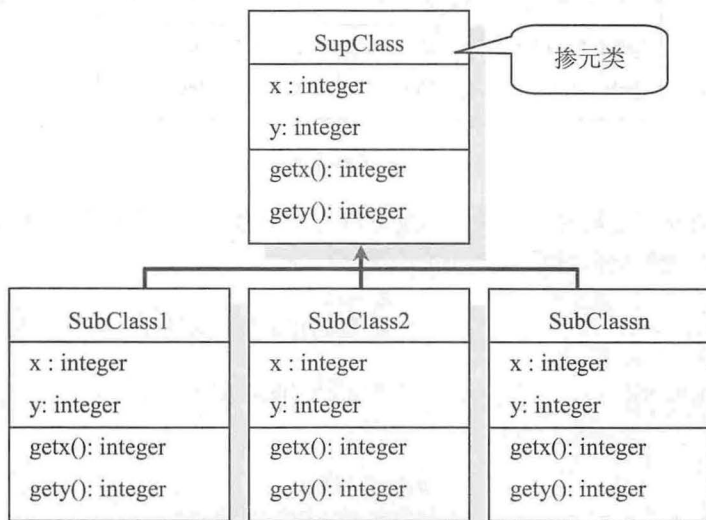


图 17.8 多亲继承示意图

如果希望某个函数被多个类调用，可以通过扩充的方式让这些类共享该函数。具体的设计思路是：先创建包含通用函数的超类，然后利用这个超类扩充子类，这种包含通用方法的类就可以称为掺元类。

例如，先设计一个掺元类 F，设想定义两个子类 A 和 B，希望子类 A 和 B 能够继承掺元类 F 的通用方法 getx() 和 gety()。代码如下：

```
var F = function(x,y){           //构造函数 F(), 掺元类
    this.x = x;                  //本地属性 x
    this.y = y;                  //本地属性 y
}
F.prototype = {                  //掺元类的原型对象
    getx : function(){           //原型方法 getx()
        return this.x;
    },
    gety : function(y){          //原型方法 gety()
        return this.y;
    }
}
```

然后，定义两个子类 A 和 B，利用类继承方法先继承掺元类中的本地属性，以方便继承的方法正确获取值。在实际应用中不用类继承来继承掺元类的本地属性和方法。

```

A = function(x,y){                                //子类 A
    F.call(this,x,y)                              //类继承, 把掺元类 F 的本地属性继承过来, 并传递参数
};
B = function(x,y){                                //子类 B
    F.call(this,x,y)                              //类继承, 把掺元类 F 的本地属性继承过来, 并传递参数
};

```

如果让 A 类和 B 类都继承 F 类, 可以使用原型继承方法来实现, 但是原型继承需要实例化类 F。可以模仿复制继承方法设计一个专门函数来实现这种继承关系, 具体代码如下:

```

//掺元类继承封装函数, 其中参数 Sub 表示子类, 参数 Sup 表示掺元类
function extend(Sub,Sup){
    for(m in Sup.prototype){                      //遍历掺元类的原型对象
        if(!Sub.prototype[m]){                  //如果子类没有存在同名成员
            Sub.prototype[m] = Sup.prototype[m]; //则复制掺元类原型成员给子类原型对象
        }
    }
}

```

该函数很简单, 使用 for/in 循环遍历掺元类的原型对象中的每一个成员, 并将其添加到子类的原型对象中。如果子类中已存在同名成员, 则跳过该成员, 转而处理下一个, 这样能够确保子类原型对象中的成员不会被改写。

有了这个封装函数, 就可以直接调用来快速生成多个相同的子类。传递子类参数必须事先声明, 且应通过类继承方法, 继承 F 的本地属性和方法。

```

extend(A,F);                                     //继承 F 的子类 A
extend(B,F);                                     //继承 F 的子类 B
最后, 实例化 A 和 B, 就可以调用 F 定义的通用方法。
var a = new A(1,2);                             //实例化 A
var b = new B(10,20);                           //实例化 B
alert(a.getx());                                //返回 1
alert(a.gety());                                //返回 2
alert(b.getx());                                //返回 10
alert(b.gety());                                //返回 20

```

也可以利用这种方法, 把多个子类合并到一个类中, 实现多重继承。例如, 下面的示例定义了两个类 A 和 B, 并分别为它们定义两个原型方法:

```

var A = function(){}                             //类 A
A.prototype = {                                  //类 A 的原型对象
    x : function(){                              //原型方法 x()
        return "x";
    }
}
var B = function(){}                             //类 B
B.prototype = {                                  //类 B 的原型对象
    y : function(){                              //原型方法 y()
        return "y";
    }
}
C = function(){}                                 //空类 C
extend(C,A);                                     //把类 A 继承给类 C
extend(C,B);                                     //把类 B 继承给类 C
var c = new C();                                 //实例化类 C
alert(c.x());                                   //返回字符 x
alert(c.y());                                   //返回字符 y

```


面向对象中并不是所有的事物泛型都是使用继承关系来描述的，继承关系只是泛型关系的一种。除此之外，创建关系、原型关系、聚合关系、组合关系等都是泛型的一种类型。泛型概念很宽泛，通常使用继承、聚合和组合来描述事物的名词特性，而使用原型、元类等其他概念来描述事物的形容词概念。

17.11 封装

封装（Encapsulation）就是把对象内部数据和操作细节进行隐藏。很多面向对象语言都支持封装特性，提供关键字（如 `private`）来隐藏某些属性和方法。要想访问被封装对象中的数据，只能使用对象专门提供的对外接口，这个接口一般为方法，调用该方法能够获取对象内部数据。

JavaScript 语言没有提供专门的信息封装关键字，不过可以使用闭包来创建，只允许从对象内部访问的方法和属性。另外，接口也是数据封装的一种工具，接口提供了外界访问方法的约定。在应用开发中，所有类都应定义接口，类只向外界提供已实现接口中规定的方法，任何别的方法都是隐藏的。其所有属性都是私有的，外界只能通过接口中定义的存取操作与之打交道。

17.11.1 被动封装

所谓被动封装，就是对对象内部数据进行适当约定，这种约定具有很强的主观性，没有强制性保证，它主要针对公共对象而言。一般来说，JavaScript 类对象所包含的数据都是公开的，没有隐私可言，任何人都可以访问其中的信息。例如：

```
var Card = function(name,sex,work,detail){    //公共类
    if(name == undefined) throw new Error("name 是必需的"){//为空则抛出错误
        this.name = name;
    }
    this.sex = sex || "男";                    //为空，则输入默认值“男”
    this.work = work;                          //没有限制
    this.detail = detail;                      //没有限制
}
```

为了数据安全，代码中适当增加了一些条件限制，避免非法信息侵入。还可以增加更完善的监测方法，以保护输入数据的完整性。代码如下：

```
var Card = function( name, sex, work, detail ){ //较安全的公共类
    if( ! checkName( name ) ) throw new Error( "name 值非法" ){
        this.name = name;
    }
    this.sex = checkSex( sex );
    this.work = checkWork( work );
    this.detail = checkDetail( detail );
}
Card.prototype = {
    checkName : function( name ){                //类内部数据检测方法
                                                //检测 name，参数为 name，返回布尔值，是否合法
    }
    checkSex : function( sex ){                  //检测 sex，参数为 sex，返回 sex，检测是否符合约定
    }
    checkWork : function( work ){                //检测 work，参数为 work，返回 work，检测是否符合约定
    }
    checkDetail : function( detail ){            //检测 detail，参数为 detail，返回 detail，检测是否符合约定
    }
}
```

上面的代码仅列出了各种方法的框架，可以根据需要定义具体检测的逻辑。从更安全和扩展的角度来讲，凡是类都应该定义接口，这样能够确保数据存取更加安全，同时也方便与其他开发人员和用户进行交流。具体方法可以参阅上面章节的讲解，这里不再重复。

内部私有方法监测和接口措施能够在一定程度上保护对象内部数据，但是它们也存在一个致命的漏洞，因为这些属性和方法可以被公开重置，面对公开覆盖属性和方法值，任何人都无法阻止这种行为。不管是有意还是无意操作，属性都可能会被设置为无效值。同时内部检测和接口在一定程度上占用了系统开销，这个问题也是必须认真考虑的。

很多开发人员习惯使用命名规范来区别公共与私有成员，就是在一些方法和属性的名称前后加下划线以示其私有特性。例如：

```
Card.prototype = {
    _checkName : function( name ){           //类内部数据检测方法
                                                //检测 name，参数为 name，返回布尔值，是否合法
    }
    _checkSex : function( sex ){             //检测 sex，参数为 sex，返回 sex，检测是否符合约定
    }
    _checkWork : function( work ){           //检测 work，参数为 work，返回 work，检测是否符合约定
    }
    _checkDetail : function( detail ){       //检测 detail，参数为 detail，返回 detail，检测是否符合约定
    }
}
```

下划线命名法是一种约定俗成的命名规范，它表明一个属性和方法仅供对象内部使用，直接访问可能会导致意想不到的后果。虽然它不是强制性规定，但是能够有助于防止开发人员无意误用。

上述数据保护的方法和措施都是被动性防御，因为它们只是一种约定，只有在得到遵守时才有效果，而且并没有什么强制性手段可以保证实施，所以也不是真正可以用来隐藏对象内部数据的解决方案。它主要适用于非敏感性的内部方法和属性。

17.11.2 主动封装

在 JavaScript 中，只有函数具有作用域。在函数内部声明的变量，在函数外部是无权访问的。从本质上分析，所谓私有属性和私有方法，就是在对象外部无法访问。所以要真正实现类的封装设计要求，使用函数作用域是最佳选择。例如，在下面的示例中包含在函数 f() 中的变量 n 和函数 e() 都不能够被外界访问，但函数 e() 可以访问 n：

```
function f(){                                //外部函数
    var n = 1;                               //私有变量
    function e(){                             //私有函数
        n++;
    };
    e();                                     //内部调用私有函数
    return n;                               //返回私有变量 n 的值
}
```

如果函数 f() 返回函数 e()，则外界是可以访问它的。例如：

```
function f(){                                //外部函数
    var n = 1;                               //私有变量
    function e(){                             //私有函数
        return ++n;
    };
    e();                                     //内部调用私有函数
    return e;                               //返回私有函数
}
```

```

}
var e = f();           //获取私有函数
alert(e());           //外部调用私有函数

```

可以看到，在上面的示例中，外界可以访问函数 `f()` 内部的私有函数 `e()`，这主要是因为 JavaScript 中的作用域是词法性质的，函数总是运行在定义它们的作用域中，而不是运行在调用它们的作用域中。

那么，根据函数的这一特性，可以把 17.11.1 节示例中的私有数据使用函数作用域和闭包进行封装。具体方法是：在函数结构体内部定义变量，这些变量可以被定义该作用域中的所有函数访问。具体代码如下：

```

var Card = function( name, sex, work, detail ){           //安全的类
    var _name = name, _sex = sex, _work = work, _detail = detail; //私有属性
    function _checkName( _name ){                         //私有方法
    }
    function _checkSex( _sex ){                           //私有方法
    }
    function _checkWork( _work ){                         //私有方法
    }
    function _checkDetail( _detail ){                     //私有方法
    }
    if( !_checkName( _name ) ) throw new Error( "name 值非法" );
        this.name = _name;
    }
    this.sex = _checkSex( _sex );
    this.work = _checkWork( _work );
    this.detail = _checkDetail( _detail );
}
Card.prototype = {
    //公共方法
}

```

如果希望外界可以访问某些私有方法，可以采用如下方法来实现：

```

var Card = function( name, sex, work, detail ){
    var _name = name, _sex = sex, _work = work, _detail = detail;
    function _checkName( _name ){
    }
    this.checkName = function(){                           //返回私有方法，实现外部调用
        return _checkName
    }
}

```

函数作用域内部的方法无权被外界访问，但是在函数作用域内的其他公共方法却可以访问它们，于是利用公共方法为中转平台，可以巧妙地把内部私有方法公开化。因此这些公共方法也被称为特权方法，即在方法的前面加上关键字 `this`。因为这些方法定义于函数作用域中，所以它们能够访问到私有属性，对于不需要直接访问私有属性和方法的方法建议放在类的原型对象中进行声明。使用这种方式创建的对象具有真正的封装特性，但它也有如下一些缺点：

- ☑ 生成的每一个新实例对象都会为每一个私有方法和特权方法生成一个新的副本。这会占用大量的系统资源，所以不适宜大量使用，仅在必要时适当使用。
- ☑ 这种方法不利于类的继承，因为所有派生的子类都不能访问超类的任何私有属性和方法。例如：

```

var Card = function(){           //超类
    var _name = 1;               //私有属性
    function _checkName(){       //私有方法
        return _name;
    }
}

```

```

    this.checkName = function(){           //公共方法
        return _name;                     //可以访问私有属性
    }
}
function F(){                             //子类
    Card.call(this);                     //使用类继承方法继承类 Card
    this.name = _name;                   //访问超类中的私有属性，抛出解析错误
}
var a = new F();                          //实例化子类
alert(a.name);                           //访问无效
alert(a._checkName());                   //无法访问，抛出解析错误

```

不过，可以通过特权方法来访问超类中的私有属性和方法：

```

alert(a.checkName());                    //访问超类公共方法，间接访问私有属性和方法

```

17.11.3 静态方法

在面向对象的编程中，类是不能够直接访问的，必须实例化后才可以访问。也就是说，大多数方法和属性与类的实例产生联系。但是静态属性和方法与类本身直接联系，可以直接从类访问，也就是说静态成员是在类上操作，而不是在实例上操作。JavaScript 核心对象中的 Math 和 Global 都是静态对象，不需要实例化，就可以直接访问。

类的静态成员（属性和方法）包括私有和公共两种类型。不管是公共还是私有，它们在系统中只有一份副本，也就是说它们不会被分成多份传递给不同的对象，而是通过函数指针进行引用，这与闭包截然不同。例如：

```

var F = (function(){                     //把闭包体（外层函数）赋值给变量 F，返回一个构造函数（内层函数）
    var _a = 1;                          //闭包体的私有变量
    this.a = _a;                          //闭包体内公共属性 a
    this.get1 = function(){               //闭包体内公共方法 get1()
        return _a;
    };
    this.set1 = function(x){              //闭包体内公共方法 set1()
        _a = x;
    };
    return function(){                   //返回的构造函数类
        this.get2 = function(){           //返回的公共方法 get2(), 可以访问私有变量
            return _a;
        };
        this.set2 = function(x){          //返回的公共方法 set2(x), 可以访问私有变量
            _a = x;
        };
    }
})();                                     //执行闭包体，返回匿名构造函数结构
//定义类的静态公共方法和属性
F.get3 = function(){                     //静态公共方法 get3(), 返回对公共方法 get1 ()的调用
    return get1();
};
F.set3 = function(x){                    //静态公共方法 set3(), 返回对公共方法 set1 (x)的调用
    set1(x);
}
F.a = a;                                 //静态公共私有属性 a

```

与一般类的创建方法一样，这里的私有成员和特权成员仍然被声明在构造器（即构造函数）中，并借

助 `var` 和 `this` 关键字来实现。但构造器却由原来的普通函数变成了一个内嵌函数，并且作为外层函数的返回值赋值给了变量 `F`，这就创建了一个闭包。在这个闭包中，还可以声明静态私有成员。例如：

```
var F = (function(){
    function set5(x){                //静态私有方法
        _a = x;
    }
    function get5(){                //静态私有方法
        return _a;
    }
})();
```

这些静态私有成员可以在构造器内部访问，这意味着所有私有函数和特权函数都能访问它们。与其他方法相比，静态方法有一个优点，那就是在内存中仅存放一份。但是那些被声明在构造器之外的公共静态方法，以及下文中将要提到的 `F` 类原型属性都不能访问在构造器中定义的任何私有属性，所以它们不是特权成员。

定义在构造器中的私有方法能够调用其中的静态私有方法，反之则不然。要判断一个私有方法是否应该被设计为静态方法，可以看它是否需要访问任何实例数据。如果不需要，那么将其设计为静态方法会更有效率，因为它只被创建一份。

类的静态公共方法和属性一般在类的外面进行定义，这种外挂定义的方式在前面的示例中也曾经介绍过。这种外挂的静态方法和属性可以直接进行访问，实际上相当于把构造器作为命名空间来使用。同时，由于它们仍然属于构造器结构的一部分，所以在这些静态方法和属性中可以访问闭包中的私有成员。例如：

```
alert(F.get3());                //直接访问类 F 的静态方法 get3(), 返回 1
alert(F.a);                    //直接访问类 F 的静态属性 a, 返回 1
F.set3(2);                    //直接访问类 F 的静态方法 set3(), 并传递值为 2, 修改私有变量的值
alert(F.get3());                //返回 2, 说明修改成功
```

位于外层函数声明之后的小括号很重要，它在代码载入之后立即执行这个函数，而不是在调用构造函数 `F` 时执行。这个函数的返回值是另一个函数，它被赋给 `F` 变量，`F` 因此成了一个构造函数。在实例化 `F` 时，所调用的是这个内层函数。外层函数仅用来创建一个可以存储静态私有成员的闭包。

类 `F` 是返回的内层函数，该值是一个构造函数，它无法访问外层函数的公共方法 `get1()` 和 `set1()`，但是能够访问返回构造函数体内的公共方法 `get2()` 和 `set2()`。例如：

```
var a = new F()                //实例化类 F
alert(a.get2());                //调用类 F 的公共方法 get2(), 返回 1
a.set2(2);                    //调用类 F 的公共方法 set2(), 修改私有变量 _a
alert(a.get2());                //调用类 F 的公共方法 get2(), 返回 2
```

但是下面的用法都是错误的。因为闭包体内的变量、属性和方法，对于级别比较低的 `F` 类来说是无权访问的。

```
var a = new F()
alert(a.get1());
a.set1(2);
alert(a.get1());
```

对于闭包体内的所有对象都可以访问闭包体内的私有或公共变量、属性和方法。由于类 `F` 是闭包体内返回的构造函数，根据作用域链，它们可以向上访问闭包所有成员。

```
F.prototype = {                //类 F 的原型对象
    get4 : function(){          //原型方法 get4()
        return get1();          //访问闭包内数据
    },
    set4 : function(x){          //原型方法 set4()
        set1(x);                //访问闭包内数据
    }
}
```

```
};
var a = new F();           //实例化类 F
alert(a.get4());          //返回 1
```

通过上面的示例，利用闭包体也可以实现对数据的封装，而且这种封装是非常有效且牢固的。

17.12 重载和多态

类是一个很抽象的概念，它包含很多特性，但 JavaScript 不支持类，所以这些特性都只能通过间接方式实现。下面就多态、多重构造和析构进行简单的说明。

17.12.1 重载

重载和覆盖是两个不同的技术概念。重载（Overload）就是同名方法可以有多个实现，它们依靠参数的类型或参数的个数来区分和识别。在 JavaScript 中，函数的参数是没有类型的，并且参数个数也是任意的。例如：

```
function f(x,y){           //无法实现重载的函数
    return x+y;
}
```

上面示例中的函数 f() 虽然指定了两个形参，但是可以在调用时传递任意多个实参，参数的类型也是任意的。由于 JavaScript 语言是弱类型语言，它不会根据传递的参数个数和类型来决定要执行的行为。因此，要定义重载方法，只能通过函数的 arguments 属性来实现。例如：

```
function f(){              //定义能够重载的求和函数
    var sum = 0;
    for( var i = 0; i < arguments.length; i ++ ){
        if(typeof arguments[i] == "number")
            sum += arguments[i];
    }
    return sum;
}
```

上面函数实现了任意多个参数求和函数的重载。不管函数 f() 中包含多少个参数，也不管参数类型如何，该函数将会自动把其中的数值类型的参数相加并返回总数。例如：

```
alert( f( 3, 4, 6, 7, 8, 9 ) ); //重载函数 f(), 返回 37
alert( f( 3, 4 ) );             //重载函数 f(), 返回 7
```

另外，结合 instanceof 运算符和 constructor 属性来判断每个参数的类型，以决定根据参数个数和类型执行不同的操作，实现更复杂的方法重载。

17.12.2 覆盖

覆盖（Overrid）是子类中定义的方法，与超类中方法同名，且参数类型和个数也相同，当子类被实例化之后，从超类中继承的同名方法将被隐藏。例如：

```
function A(){              //超类 A
    this.m = function(){   //超类的本地方法 m()
        alert("A");
    }
}
function B(){              //子类 B
```

```

    this.m = function(){                //子类的本地方法 m()
        alert("B");
    };
}
B.prototype = new A();                //通过原型继承方法，类 B 继承类 A
B.prototype.constructor = B;          //恢复 B 类的原型对象的构造器
var b = new B();                      //实例化 B
b.m();                                //返回字符 B，说明子类 B 的方法 m()将覆盖 A 的方法 m()

```

当超类同名方法被覆盖之后，在强类型语言中，覆盖的方法里面可以调用被覆盖超类的方法。不过在 JavaScript 中可以通过临时私有变量先保存超类的同名方法，然后在子类同名方法中调用即可。例如：

```

function A(){                          //超类 A
    this.m = function(){               //超类的本地方法 m()
        alert("A");
    }
}
function B(){                          //子类 B
    var m = this.m;                   //先使用私有变量保存超类继承来的同名方法
    this.m = function(){               //子类的本地方法 m()
        m.call(this);                 //在当前对象中调用子类私有变量中存储的超类方法 m()
        alert("B");
    };
}
B.prototype = new A();                //通过原型继承方法，类 B 继承类 A
B.prototype.constructor = B;          //恢复 B 类的原型对象的构造器
var b = new B();                      //实例化 B
b.m();                                //返回字符 B，说明子类 B 的方法 m()将覆盖 A 的方法 m()

```

在覆盖方法中调用超类同名方法时，需要使用 `call()` 或 `apply()` 方法来改变，执行上下文为 `this`（即当前对象）。如果直接调用该方法（即 `m()`），执行上下文就会变成全局对象，在特殊语境中可能会发生相互影响。

17.12.3 多态

加号运算符就是一个典型的多态模式。例如：

```

function F(x,y){                      //构造函数
    this.x = x;
    this.y = y;
}
F.prototype.add = function(){         //原型方法，求和
    return this.x + this.y;
}
var f1 = new F(1,2);                  //传入的是数值
var f2 = new F("1","2");              //传入的是字符串
alert(f1.add());                      //返回 3，数值相加
alert(f2.add());                      //返回字符串 12，字符串相连接

```

在 JavaScript 中，加号就是一个多态运算符，它能够根据传入值的类型执行不同的计算。从某种意义上来说，多态是面向对象中重要的一部分，也是实施继承的主要目的。一个实例可以拥有多个类型，它即可以是这种类型，也可以是那种类型，这种多种状态被称为类的多态。

多态表现为两个方面：类型的模糊和类型的识别。JavaScript 是一种弱类型语言，通过 `typeof` 运算符来判断值的类型，但是它无法确定对象的类型，所有类型的实例对象对于 `typeof` 运算符来说都是基本的 `object`，因此 JavaScript 的类型是比较模糊的。由于没有严格的类型检测，因此可以为任何对象调用任何方法，而无

须考虑它是否被设计为拥有该方法。而使用 JavaScript 的原型可以设计类的多态特性。例如：

```
function A(){                                //超类 A
    this.get = function(){                  //类 A 的本地方法 get()
        alert("A");
    }
}
function B(){                                //子类 B
    this.get = function(){                  //类 B 的本地方法 get()
        alert("B");
    }
}
B.prototype = new A();                      //使用原型继承法，设置 B 类继承 A 类
function C(){                                //子类 C
    this.get = function(){
        alert("C");
    }
}
C.prototype = new A();                      //使用原型继承法，设置 C 类继承 A 类
function F(x){                              //多态类 F
    this.x = x;                            //本地属性 x
}
F.prototype.get = function(){               //多态类 F 的原型方法 get()
    if(this.x instanceof A)                //判断是否为超类的实例，然后调用不同类的方法
        this.x.get()
}
var b = new B();                            //实例化 B
var c = new C();                            //实例化 C
var f1 = new F(b);                          //把实例 b 传递给 F 进行实例化
var f2 = new F(c);                          //把实例 c 传递给 F 进行实例化
f1.get();                                   //返回 B，此时该方法指向的是 B 类中的方法 get()
f2.get();                                   //返回 C，此时该方法指向的是 C 类中的方法 get()
```

在上面的示例中，类 F 就包含了一个多态方法 get()，它能够根据不同实例执行不同的方法。

17.13 构造和析构

构造和析构是创建和销毁对象的过程，它们是对象生命周期中的起点和终点，也是最重要的环节。当创建对象时，构造函数负责创建并初始化对象的内部环境，包括分配内存、创建内部对象和打开相关的外部资源等。而当注销对象时，析构函数负责关闭资源、释放内部的对象和已分配的内存。

17.13.1 构造

在面向对象的编程中，构造和析构是类的两个重要特性。构造函数将在对象产生时调用，析构函数将在对象销毁时被调用。调用的过程和实现方法由编译器完成，只要记住它们调用的时间就行了，而且它们的调用是自动完成的。

在 JavaScript 中，被 new 运算符调用的函数就是构造函数。构造函数被 new 运算符计算之后，将返回实例对象，也就是所谓的对象初始化，即对象的诞生。构造函数实际上就是前面所说的类，调用构造函数的过程也是类实例化的过程。例如，构造函数没有返回值，可以使用 this 关键字指定对象的属性和方法等：


```
function F(x,y){           //构造函数
    this.x = x;
    this.y = y;
}
```

JavaScript 为每个对象都定义了 `constructor` 属性，该属性值指向构造函数本身。例如：

```
var f = new F(1,2);        //初始化对象，
alert(f.constructor == F); //返回 true，说明 F 是 f 对象的构造函数
```

如果构造函数有返回值，且返回值是引用类型的，那么经过 `new` 运算符计算之后，返回的不再是构造函数自身对应的实例对象，而是构造函数包含的返回值。

```
function F(x,y){           //构造函数
    this.x = x;
    this.y = y;
    return [];              //返回空的数组直接量
}
var f = new F(1,2);        //实例化对象
alert(f.constructor == F); //返回 false，说明 F 不再是 f 的构造函数
```

在上面的示例中，实例化构造函数之后返回值是一个空的数组直接量，而不再是实例对象。原来构造函数的返回值覆盖了 `new` 运算符的运算结果，此时如果调用 `f` 的 `constructor` 属性，返回值如下：

```
function Array () {        //返回被封闭的 Array 核心结构
    [ native code ]
}
```

说明它是 `Array` 的实例，使用下面的代码可以检测出来：

```
alert(f.constructor == Array); //返回 true，说明 Array 是 f 的构造函数
```

根据语约定，构造函数是没有返回值的。在构造函数中定义 `return` 语句，可以改变对象，然后修改对象的 `constructor` 属性，通过修改能够改变对象的构造器。例如：

```
f.constructor = F;         //改变对象 f 的构造器
alert(f.constructor == F); //返回 true，说明修改成功
alert(f.constructor == Array); //返回 false，说明 f 不再是 Array 的实例
```

在构造对象的过程中，如果利用 `call()` 和 `apply()` 方法，还可以实现动态构造，从而实现更加灵活的设计。例如，在下面这个示例中，构造函数 `A`、`B` 和 `C` 相互之间通过 `call()` 方法关联在一起。当构造对象 `c` 时，将调用构造函数 `C`；而在执行构造函数 `C` 中，会先调用构造函数 `B`。调用构造函数 `B` 之前，会自动调用构造函数 `C`，从而实现动态构造对象的效果。这种多个构造函数相互关联在一起，也有人称之为多重构造。不过，它们的本质显示了构造对象的动态性。代码如下：

```
function A(x){              //构造函数 A()
    this.x = x || 0;
}
function B(x){              //构造函数 B()
    A.call(this,x);         //动态构造 A
    this.a = [x];
}
function C(x){              //构造函数 C()
    B.call(this,x);         //动态构造 B
    this.y = function(){
        return this.x;
    }
}
var c = new C(3);           //实例化构造函数 C()
alert(c.y());              //返回 3
```

根据动态构造的这种特性，还可以设计类的多态处理。例如：

```

function F( x, y ){
    function A( x, y ){
        this.add = function(){
            return x + "" + y;
        }
    }
    function B( x, y ){
        this.add = function(){
            return x + y;
        }
    }
    if( typeof x == "string" || typeof y == "string" ){
        A.call( this, x, y );
    }
    else{
        B.call( this, x, y );
    }
}
var f1 = new F( 3,4 );
alert( f1.add() );
var f2 = new F( "3","4" );
alert( f2.add() );

```

//多态类型
//包含类 A，计算两个字符串相连接
//字符串连接的方法
//包含类 B，计算两个数值相加
//数值相加的方法
//如果参数中包含有字符串型数值
//则动态构造类 A，让类 F 继承 A 的方法
//否则，动态构造类 B，让类 F 继承 B 的方法
//实例化类 F，传递数值
//调用对象方法 add()，返回数值 7
//实例化类 F，传递字符串
//调用对象方法 add()，返回字符串 34

17.13.2 析构

析构是销毁对象的过程。由于 JavaScript 能够自动回收垃圾，不需要人工清除，所以当对象使用完毕时，JavaScript 会调用对象回收程序来销毁内存中的对象，这个回收程序相当于一个析构函数。JavaScript 是不支持析构语法的。在下面示例中，先定义一个析构函数，该函数中包含一个析构方法，把该方法继承给任意对象，就可以调用它清除对象内部所有成员。代码如下：

```

function D(){
}
D.prototype = {
    d : function(){
        for( var i in this ){
            if( this[i] instanceof D ){
                this[i].d();
            }
            this[i] = null;
        }
    }
}
function F(){
    this.x = 1;
    this.y = function(){
        alert( 2 );
    }
}
F.prototype = new D();
var f = new F();
f.d();
alert(f.x);
f.y()

```

//析构函数
//析构函数原型对象
//析构方法
//遍历当前对象
//如果发现包含析构函数的实例
//则递归调用析构方法
//设置对象成员的值为 null，即清除成员
//试验函数
//绑定析构函数，继承析构方法
//实例化试验函数
//调用析构方法
//返回 null，说明属性已经被注销
//返回编译错误，说明方法已不存在

在强类型语言中，构造是从基类开始按继承的层次顺序调用，析构的时候顺序正好相反。这样处理是因为子类可能在构造函数里使用父类的成员变量，如果父类还没有创建，那就会有问题。而析构的时候，如果父类先析构，也出现这样的问题。JavaScript 对此没有严格的要求，但是它遵循从下到上的顺序来进行构造，而析构没有这方面的要求，只要对象没有成员引用或对象引用即可。

17.14 扩展

在面向对象的语言中，类是面向对象的基础，同时类具有明显的层次概念和继承关系，每个类都有一个超类，它们从超类中继承属性和方法。类还可以进一步地被扩展，扩展类被称为子类，这样就构建了一个多层、复杂的对象继承关系。但是 JavaScript 是基于对象的语言，它是以对象为基础，以函数为模型，以原型为继承机制的开发模式。

17.14.1 超类和子类

在 JavaScript 中，Object 对象是通用类，其他所有内置对象和自定义的构造对象都是专用类。也就是说，Object 对象是超类，而其他内置对象和自定义构造对象都是 Object 对象的子类。因此，在 JavaScript 中所有对象都拥有 Object 对象定义的属性和方法。

JavaScript 语言的继承机制是通过原型继承来实现的，而不是通过类继承来实现的。而原型对象本身又是一个实例对象，它是由构造函数 Object() 创建的。因此，所有原型对象都继承了 Object.prototype 属性。例如，为 Object 对象的原型对象定义一个属性 name，则内置对象 Date 和自定义构造对象 Me 都自动继承了该原型属性。

```
Object.prototype.name = "Object 对象的原型属性";    //超类的原型属性
var d = Date.prototype;                             //内置对象的原型对象
alert(d.name);                                       //返回超类的原型属性值
function Me(){}                                     //自定义类
}
var m = Me.prototype;                               //自定义类的原型对象
alert(m.name);                                       //返回超类的原型属性值
```

由于内置对象 Date 和自定义类(或构造对象)Me 通过原型机制继承了 Prototype 对象的属性，而 Prototype 对象又继承了 Object.prototype 的属性，因此 Date 和 Me 子类实际上继承了两个原型对象的属性。在解析时，JavaScript 解释器首先在实例对象中查询属性 name。如果没有发现要查询的属性，则会顺着原型链，查询子类包含的 Prototype 对象。如果在子类的原型对象中没有找到属性 name，就会查询超类 Object 的 Prototype 对象，最终读取 Object.prototype 对象定义的属性值。

在 JavaScript 中，所有类都是 Object 对象的子类，JavaScript 仅支持两层关系的类结构，这主要是因为 JavaScript 是由一种脚本语言的特性决定的，也可以设计更复杂的多层类结构。例如，在下面的示例中，构建了 3 层类结构体系。其中超类 Object 属于顶层，而类 Me 属于 Object 超类的子类，同时它也是 Sub 类的父类，或者说 Sub 类是 Me 的子类。

```
Object.prototype.name = "超类的原型属性";           //超类的原型属性
function Me(){}                                     //父类
    this.saying =function(){                         //父类的方法
        return "父类的方法"
    };
}
function Sub(){}                                    //子类
}
```

```

Sub.prototype = new Me();           //建立类的多层继承关系
var m = new Sub();                 //实例化子类
alert(m.name);                     //继承了超类的原型属性 name
alert(m.saying());                 //继承了父类的方法 saying()

```

在默认情况下，类的原型对象的构造器应该是类本身，`prototype.constructor` 属性总是指向类自身。但是，如果把父类的实例传递给子类的 `prototype` 属性，就会破坏了原型对象与默认构造器的引用关系，而从指向父类实例的构造器了，这样就容易引发构造器关系的混乱。例如，在正常情况下，子类原型对象的构造器会指向子类自身。

```

function Sub(){
    alert(Sub.prototype.constructor); //指向 Sub 类的引用
}
而下面的做法就破坏了这种关系：
function Sub(){
    Sub.prototype = new Me();
    alert(Sub.prototype.constructor); //指向 Me 类的引用
}
为了纠正这个错误，可以手动修改子类原型对象的构造器引用：
function Sub(){
    Sub.prototype = new Me();
    Sub.prototype.constructor = Sub; //修改子类原型对象的构造器为默认值，即引用子类自身
    alert(Sub.prototype.constructor); //指向 Sub 类的引用
}

```

17.14.2 元类

元类就是类的类型，即创建类型的类。元类与类的关系正如类与对象的关系一样，是一种创建型的泛化关系。元类能够接收类作为参数的类，即元类操作的对象是类，而不是具体的数据。一般元类返回的是类，而不是具体的数据。例如，下面的示例就是一个简单的元类，它包含了一个返回的类。

```

function O(x){                       //元类
    return function(){               //返回类
        this.x = x;                 //返回类的属性
        this.get = function(){      //返回类的方法
            alert(this.x);
        }
    }
}
var o = new O(1);                     //实例化元类
var f = new o();                      //实例化元类返回的类
f.get();                             //调用返回类的方法 get(), 返回 1

```

元类与普通函数没有什么两样，不过它的返回值是类，而不是具体数值。实际上，JavaScript 核心对象 `Function` 就是一个元类，虽然它没有返回值，但是可以通过字符串的形式创建返回类。例如：

```

var O = new Function("this.x=1;this.y=2") //实例化之后返回的是类
var o = new O();                          //实例化返回类
alert(o.x);                              //调用实例的属性值，返回 1

```

上面示例的函数中包含一个返回类结构，当然元类并非如此简单。下面再演示一个比较复杂的示例，在这个元类中参数值包含类类型，返回值也是类类型。首先，定义一个普通类，作为一个参数值准备传递给元类：

```

function F(x, y){                    //定义作为参数的类
    this.x = x;
    this.y = y;
}
F.prototype.add = function(){        //类的原型方法

```



```

    alert( this.x + this.y );
}

```

然后，定义一个元类，该函数类包含 3 个参数。其中第一个参数为类类型，第 2、3 个参数是值类型数据：

```

function O( o, x, y ){
    this.say = function(){
        alert( "元类" );
    }
    return function(){
        this.say = function(){
            alert( "返回类" );
        }
        var a = new o( x, y );
        for( var i in a ){
            this[i] = a[i];
        }
    }
}

```

//创建元类，第 1 个参数为类类型，第 2、3 个参数为数值
//元类的本地方法
//返回类
//返回类的本地方法
//实例化参数类
//通过实例继承法，继承参数类给返回类
//此时 this 关键字指返回类的当前对象

最后，使用 new 运算符调用元类，第 1 个参数值为上文定义的类 F，第 2、3 个参数为普通数值，返回的类赋值给变量 A，则 A 就变成了一个类结构。此时不能够通过 A 来读取元类的本地方法 say()：

```

var A = new O( F, 1, 2 );
var B = new A();
A.say();

```

//实例化元类
//实例化元类返回类
//如果直接调用元类的本地方法，将提示编辑错误

通过实例化后的 B 对象来访问参数类 F 中的成员，以及返回类内部定义的本地属性：

```

B.say();
B.add();
alert( B.x );
alert( B.y );

```

//返回字符串“返回类”，调用返回类自己的本地方法
//返回数值 3，调用参数类的原型方法
//返回数值 1，调用参数类的本地属性
//返回数值 2，调用参数类的本地属性

当一个类有返回值时，如果是值类型数据，则可以访问类的成员，也可以获取返回值。例如：

```

function F(){
    this.x = 1;
    return 2;
}
var f = new F();
alert(f.x);
alert(F());

```

//类 F
//本地属性
//返回值
//实例化类 F
//返回 1，可以访问
//返回 2，可以访问

但是如果类返回的是引用类型或者是函数体，则类的成员将不可访问，它们将成为闭包结构内的私有数据，不再对外开放。例如：

```

function F(){
    this.x = 1;
    return function(){
        return this.x;
    };
}
F.prototype.y = function(){
    alert(3);
}
var f = new F();
alert(f.x);
alert(F());
alert(f.y());

```

//类 F
//本地属性
//返回函数
//可以访问本地属性值
//类的原型方法
//实例化类 F
//访问本地属性 x 失败，返回 undefined
//调用返回的函数，返回 1，说明它可以访问本地属性 x
//提示编辑错误，没有这个成员

第18章

jQuery 框架透析之实战

( 视频讲解: 1 小时 52 分钟)

jQuery 是非常优秀的 JavaScript 库, 与 Prototype、YUI、Mootools 等众多的 JavaScript 类库相比, 它剑走偏锋, 从 Web 开发实用角度出发, 抛弃了其他库中一些不实用的东西, 为开发者提供了优美、短小而精悍的类库, 极大地提高了 Web 系统的开发效率, 因此可以说是 Web 应用开发中最佳的 JavaScript 辅助类库之一。大部分开发者正在抛弃 Prototype 等类库, 而选择 jQuery 作为进行 Web 开发的 JavaScript 库。

如果开发人员仅知道 jQuery 用法简单, 却不明白 jQuery 的运行原理和内部机制, 在使用时肯定会碰到很多问题。这些问题有一部分是 jQuery 的 Bug, 但大部分是由于自身使用不当而造成的。而 jQuery 帮助文档的简单使用说明很难解决问题。在调试基于 jQuery 的 Web 应用时, 很多时候都要跟踪进入 jQuery 对象分析其运行状态以了解出错的原因。

如果对于 Web 应用的页面运行性能和效率有所要求, 那么用户更应该明白 jQuery 运行机理和核心源码。但是 jQuery 源码不像其他的类库那样, 它有点晦涩、难懂, 本章建立在前面两章 JavaScript 编程核心知识基础上, 详解 jQuery 源码, 帮助读者快速领略 jQuery 框架设计思想、思路 and 实现步骤, 并在开发中的应用。

18.1 设计思路

在使用 jQuery 之前, 读者也许会问 jQuery 是什么。

其实它的名字就表达了其设计主旨。j 是 JavaScript 的缩写, Query 是指查询, 可以把 jQuery 看作是一个查询 JavaScript 的类库。它与 Prototype、Mootools 等类库一样, 为 Web 开发提供了 JavaScript 辅助功能。

为什么要选用 jQuery 呢? 在 jQuery 出现之前, Prototype、YUI 已经是成熟的 JavaScript 框架, 且各有特点, 市场占有率比较高。为什么开发者会抛弃它们, 而使用后起之秀的 jQuery, 它有什么优秀的特性吸引开发人员呢?

回答这个问题, 需要先明白 jQuery 的设计理念。简单总结一下, 用户在 Web 开发中主要做了如下 5 方面的事情:

- ☑ 使用 `getElementById()` 和 `getElementsByName()` 方法在 DOM 文档中找到元素, 然后取值或设置。
- ☑ 使用 `innerHTML` 属性读写元素包含的内容。
- ☑ 监听元素事件, 如 `click`。
- ☑ 通过改变元素的 CSS 样式, 实现各种视觉效果, 如高度、宽度、位置、透明度、背景色、边框等样式。

- ☑ 通过 Ajax 从服务器取值，并在指定元素里显示内容。

因此，在使用 JavaScript 开发的时候就是在对 DOM 元素进行操作。这个 DOM 元素可能是单个或集合的形式。对于 Document、Window 等对象可以直接引用，但是对于其他 DOM 元素，需要从文档树查找找到它。这样就可以把 JavaScript 操作分解为以下两部分。

- ☑ 查找 DOM 元素。
- ☑ 对 DOM 元素进行操作。

对于能够熟练使用 JavaScript 的开发者而言，也许手写 `document.getElementById()` 或 `element.getElementsByTagName` 等这样冗长的代码就可以直接查找 DOM 元素。同样对于元素的样式和事件等操作，都可以借助 JavaScript 原生的方法和属性来实现。但是对于 IE、mozilla 等几大主流的浏览器的兼容足够让每一个 JavaScript 高手头疼，这也是选用各种 JavaScript 类库的主要原因。JavaScript 类库只要使用恰当，不一定比直接采用 JavaScript 的原始函数和对象的运行效率低，但是却能极大地提高开发的效率。

自从 Prototype 采用 \$ 符号作为 `document.getElementById` 的缩写，\$ 符号似乎成了查找元素的代理符号。但是这种简单的查找并不能满足 Web 应用的需要，很多时间需要像 CSS 选择器那样查找 DOM 元素。

jQuery 采用 \$ 符号作为查找元素的代理，它不再是那种简单的 `getElementById`，而是功能强大的 CSS 选择器，这也就是 jQuery 的本意。

解决了查找元素的任务，下面就对元素进行操作。jQuery 抛弃了 Prototype 中对 Array、Object、Function、Event 等 JavaScript 原生对象的扩展，把所有的焦点都放在解决实际问题的 DOM 元素的操作上。它不仅简化 DOM 元素原生方法难记、难写的问题，而且在简化这些方法的同时提供了更为便捷且兼容浏览器的功能。同时那些实用的功能一个都没少，如 Ajax、Event、Fx、CSS 的操作一应俱全。

Prototype 中 Event、Ajax 等众多的对象不但让人觉得繁琐难记，而且让人感觉有点畏惧。jQuery 在设计时就考虑到这一点，它提供了统一的入口，就是一个 jQuery 对象 (\$)，所有的操作和变化都是针对这个对象进行的。

实际上，jQuery 就是一个查询器。在查询器的基础上还提供对查找到的元素进行操作的功能。这样说来 jQuery 就是查询和操作的统一。查询是入口，操作是结果。从代码角度分析，jQuery 对象分成以下两大部分。

- ☑ jQuery 静态方法：也称为实用方法或工具方法，通过 `jQuery.xxx()` 的 jQuery 命名空间直接引用。
- ☑ jQuery 实例方法：通过 `jQuery(xx)` 或 `$(xx)` 来生成 jQuery 实例，然后通过这个实例来引用的方法。这部分方法大多数是从采用静态方法代理来完成功能。真正的功能性的操作都在 jQuery 的静态方法中实现。

这些功能细分起来，可以分成以下几个部分。

- ☑ Selector 查找元素：这个查找不但包含基于 CSS 1~CSS 3 的 CSS 选择器功能，还包含其对直接引用或间接引用 DOM 元素而扩展的一些功能。
- ☑ DOM 元素的属性操作：DOM 元素可以看作 HTML 标签，对于属性的操作就是对于标签的属性进行操作。这个属性操作包含增加、修改、删除、取值等。
- ☑ DOM 元素的 CSS 操作：CSS 用于控制页面显示的效果。对 CSS 的操作需要包含高度、宽度、显示等常用样式设计功能。
- ☑ Ajax 的操作：Ajax 的功能就是异步从服务器取数据然后进行相关操作。
- ☑ Event 的操作：对 Event 的兼容做了统一的处理。
- ☑ 动画 (Fx) 的操作：可以看作是 CSS 样式上的扩展。

18.2 设计框架

jQuery 框架源代码非常长，普通用户是无法在短时间内读懂其内部逻辑的。为了方便读者更容易理解

jQuery 框架的搭建过程，下面通过简单的模拟和拆解，讲解 jQuery 框架设计的实现过程。

18.2.1 定义构造函数

在 JavaScript 脚本中，到处都是函数，函数可以实现类，这个类是面向对象编程中的最基本概念，也是最高抽象。定义一个类就相当于制作了一个模型，然后借助这个模型复制无数的实例。

例如，下面代码就可以定义最初的 jQuery 类。类名就是 jQuery，可以把它视为一个函数，函数名是 jQuery。当然，也可以把它视为一个对象，对象名是 jQuery。

```
<script language="javascript" type="text/javascript">
var jQuery = function(){
    //函数体
}
</script>
```

上面代码创建了一个空的函数，好像什么都不能够做，这个函数实际上就是所谓的构造函数。构造函数在面向对象语言中是类的一个特殊方法，用来创建类。在 JavaScript 中，可以把任何函数都视为构造函数。

所有类都有最基本的功能，如继承、派生、重写等。JavaScript 很奇特，它通过为所有函数绑定一个 prototype 属性，由这个属性指向一个原型对象，原型对象中可以定义类的继承属性和方法等。所以，对于上面的空类，可以继续扩展原型。代码如下：

```
<script language="javascript" type="text/javascript">
var jQuery = function(){}
jQuery.prototype = {
    //扩展的原型对象
}
</script>
```

原型对象是 JavaScript 实现继承的基本机制。然后为其起个别名，如 fn。如果直接命名 fn，则表示该名称属于 Window 对象，即全局变量名。更安全的方法是为 jQuery 类定义一个公共属性 jQuery.fn，然后把 jQuery 的原型对象传递给这个公共属性。实现代码如下：

```
<script language="javascript" type="text/javascript">
jQuery.fn = jQuery.prototype = {
    //扩展的原型对象
}
</script>
```

这里的 jQuery.fn 相当于 jQuery.prototype 的别名，为了方便以后使用，它们指向同一个引用。因此要调用 jQuery 的原型方法，直接使用 jQuery.fn 公共属性即可，不用直接引用 jQuery.prototype。当然直接使用 jQuery.prototype 也是可以的。

既然原型对象可以使用别名，jQuery 类也可以起个别名，可以使用 \$ 符号来引用它：

```
var $ = jQuery = function(){}
现在模仿 jQuery 框架源码，给它添加两个成员，一个是原型属性 jQuery，一个是原型方法 size():
```

```
<script language="javascript" type="text/javascript">
var $ = jQuery = function(){}
jQuery.fn = jQuery.prototype = {
    jQuery: "1.7.2", //原型属性
    size: function() { //原型方法
        return this.length;
    }
}
</script>
```


18.2.2 返回 jQuery 对象

针对 18.2.1 节的代码，当为 jQuery 添加 jQuery 属性和 size() 方法之后，这个框架最基本的样子就出来了。但是，该如何调用 jQuery 属性和 size() 方法呢？

也许，可以这样使用：

```
<script language="javascript" type="text/javascript">
var my$ = new $();           //实例化
alert( my$.jQuery );         //调用属性，返回 1.7.2
alert( my$.size() );         //调用方法，返回 undefined
</script>
```

但是，jQuery 好像不是这样使用的。它模仿类似下面的方法进行调用：

```
$.jQuery;
$.size();
```

也就是说，jQuery 没有使用 new 运算符实例化 jQuery 类，而是直接调用 jQuery() 函数，然后在这个函数后面直接调用 jQuery 的原型方法。

如果模仿 jQuery 框架的用法执行下面代码，浏览器会显示编译错误，这说明上面这个案例代码还不是真正的 jQuery 技术原型。

```
alert($.jQuery );
alert($.size() );
```

也就是说，把 jQuery 即看作一个类，同时也应该把它视为一个普通函数，并让这个函数的返回值为 jQuery 类的实例。因此，应该是下面这种结构模型：

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return new jQuery();    //返回类的实例
}
jQuery.fn = jQuery.prototype = {
    jQuery: "1.7.2",
    size: function() {
        return this.length;
    }
}
alert($.jQuery );
alert($.size() );
</script>
```

但是，如果在浏览器中预览，则会提示内存外溢错误，说明出现了死循环引用。如何返回一个 jQuery 实例呢？

当使用 var my\$ = new \$() 创建 jQuery 类的实例时，this 关键字就指向对象 my\$，因此，my\$ 实例对象就获得了 jQuery.prototype 包含的原型属性或方法，这些方法内的 this 关键字就会自动指向 my\$ 实例对象。

因此，可以在 jQuery 中使用一个工厂方法来创建一个实例，把这个方法放在 jQuery.prototype 原型对象中，然后在 jQuery() 函数中返回这个原型方法的调用：

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return jQuery.fn.init();    //调用原型方法 init()
}
jQuery.fn = jQuery.prototype = {
    init : function(){          //在初始化原型方法中返回实例的引用
        return this;
    }
}
```

```

    },
    jQuery: "1.7.2",
    size: function() {
        return this.length;
    }
}
alert( $.jQuery );           //调用属性, 返回 1.7.2
alert( $.size() );          //调用方法, 返回 undefined
</script>

```

18.2.3 设计作用域

下面初步实现让 jQuery() 函数能够返回 jQuery 类的实例: init() 方法返回的是 this 关键字, 该关键字引用的是 jQuery 类的实例。如果在 init() 函数中继续使用 this 关键字, 也就是说, 假设把 init() 函数也视为一个构造器, 则其中的 this 该如何理解和处理呢?

例如, 在下面示例中, jQuery 原型对象中包含一个 length 属性, 同时 init() 从一个普通的函数, 转身变成了构造器, 它也包含一个 length 属性和一个 test() 方法。从运行示例可以看到, this 关键字引用了 init() 函数作用域所在的对象。此时访问 length 属性时, 返回 0。而 this 关键字也能够访问上一级对象 jQuery.fn 对象作用域, 所以 \$.jQuery 返回 1.7.2。但是调用 \$.size() 方法时, 返回的是 0, 而不是 1。

```
<script language="javascript" type="text/javascript">
```

```
var $ = jQuery = function(){
    return jQuery.fn.init();
}
```

```
jQuery.fn = jQuery.prototype = {
    init: function(){
```

```
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
```

```
    jQuery: "1.7.2",
    length: 1,
    size: function() {
        return this.length;
    }
}
```

```
alert( $.jQuery );           //返回 1.7.2
```

```
alert( $.test() );          //返回 0
```

```
alert( $.size() );          //返回 0
```

```
</script>
```

这种设计思路很容易破坏作用域的独立性, 对于 jQuery 这样的框架来说, 很可能会造成消极影响。因此, 可以看到 jQuery 框架是通过下面方式调用 init() 初始化构造函数的:

```
<script language="javascript" type="text/javascript">
```

```
var $ = jQuery = function(){
    return new jQuery.fn.init();    //实例化 init 初始化类型, 分隔作用域
}
```

```
</script>
```

这样就可以把 init() 构造器中 this 和 jQuery.fn 对象中的 this 关键字隔离开来, 避免相互混淆。但是, 通

过这种方式也会带来另一个问题，即无法访问 jQuery.fn 对象的属性或方法。例如，在下面示例中，访问 jQuery.fn 原型对象的 jQuery 属性和 size() 方法就会出现問題。

```
<script language="javascript" type="text/javascript">
var $=jQuery = function(){
    return new jQuery.fn.init();
}
jQuery.fn = jQuery.prototype = {
    init : function(){
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
    jQuery: "1.7.2",
    length: 1,
    size: function() {
        return this.length;
    }
}
alert( $.jQuery );           //返回 undefined
alert( $.test() );          //返回 0
alert( $.size() );          //抛出异常
</script>
```

18.2.4 跨域访问

如何做到既分隔初始化构造函数与 jQuery 原型对象的作用域，又能够在返回实例中访问 jQuery 原型对象呢？

jQuery 框架巧妙地通过原型传递解决了这个问题，它把 jQuery.fn 传递给 jQuery.fn.init.prototype，也就是说，是用 jQuery 的原型对象覆盖 init 构造器的原型对象，从而实现跨域访问。代码如下：

```
<script language="javascript" type="text/javascript">
var $=jQuery = function(){
    return new jQuery.fn.init();
}
jQuery.fn = jQuery.prototype = {
    init : function(){
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
    jQuery: "1.7.2",
    length: 1,
    size: function() {
        return this.length;
    }
}
jQuery.fn.init.prototype = jQuery.fn; //使用 jQuery 的原型对象覆盖 init 的原型对象
```

```

alert( $.jQuery );           //返回 1.7.2
alert( $.test() );          //返回 0
alert( $.size() );          //返回 0
</script>

```

new jQuery.fn.init()创建的新对象拥有 init 构造器的 prototype 原型对象的方法，通过改变 prototype 指针的指向，使其指向 jQuery 类的 prototype，这样创建出来的对象就继承了 jQuery.fn 原型对象定义的方法。

18.2.5 设计选择器

jQuery 返回的是 jQuery 对象，jQuery 对象是一个类数组的对象，本质上它就是一个对象，拥有数组的长度和下标，但是没有继承数组的方法。前面几节的讲解都是建立在一种空理论上，目的是希望读者理解 jQuery 框架的核心构建过程。下面就尝试为 jQuery() 函数传递一个参数，并让它返回一个 jQuery 对象。

jQuery() 函数包含 selector 和 context 两个参数。其中，selector 表示选择器，而 context 表示选择的内容范围对象，它表示一个 DOM 元素。为了简化操作，假设选择器的类型仅限定为标签选择器。实现的代码如下：

```

<div></div>
<div></div>
<div></div>
<script language="javascript" type="text/javascript">
var $ =jQuery = function(selector, context ){           //定义类
    return new jQuery.fn.init(selector, context );     //返回选择器的实例
}
jQuery.fn = jQuery.prototype = {                      //jQuery 类的原型对象
    init : function(selector, context){               //定义选择器构造器
        selector = selector || document;              //设置默认值为 document
        context = context || document;                //设置默认值为 document
        if ( selector.nodeType ) {                   //如果选择符为节点对象
            this[0] = selector;                      //把参数节点传递给实例对象的数组
            this.length = 1;                          //并设置实例对象的 length 属性，定义包含元素个数
            this.context = selector;                  //设置实例的属性，返回选择范围
            return this;                              //返回当前实例
        }
        if ( typeof selector === "string" ) {         //如果选择符是字符串
            var e = context.getElementsByTagName(selector); //获取指定名称的元素
            for(var i = 0;i<e.length;i++){             //遍历元素集合，并把所有元素填入到当前实例数组中
                this[i] = e[i];
            }
            this.length = e.length;                    //设置实例的 length 属性，即定义包含元素的个数
            this.context = context;                    //设置实例的属性，返回选择范围
            return this;                              //返回当前实例
        } else{
            this.length = 0;                           //否则，设置实例的 length 属性值为 0
            this.context = context;                     //设置实例的属性，返回选择范围
            return this;                              //返回当前实例
        }
    },
    jQuery: "1.7.2",
    size: function() {
        return this.length;
    }
}

```



```
jQuery.fn.init.prototype = jQuery.fn;
alert( $("div").size() );           //返回 3
</script>
```

在上面示例中, `$("div")` 基本拥有了 jQuery 框架中 `$("div")` 语法的功能, 使用它可以选取页面中的指定范围的 `div` 元素。同时, 调用 `size()` 方法可以返回 jQuery 对象集合的长度。

18.2.6 设计迭代器

在 jQuery 框架中, jQuery 对象有多重身份:

- ☑ jQuery 对象是一个数据集合, 它不是一个个体对象, 无法直接使用 JavaScript 的方法来操作它。
- ☑ jQuery 对象实际上就是一个普通的对象, 因为它通过 `new` 运算符创建的一个新的实例对象。它可以继承原型方法或属性, 同时也拥有 `Object` 类型的方法和属性。
- ☑ jQuery 对象包含数组特性, 因为它赋值了数组元素, 以数组结构存储返回的数据。

因此可以以 JavaScript 的概念理解 jQuery 对象。例如:

```
<script language="javascript" type="text/javascript">
var jQuery = {                                //定义对象直接量
    name : "jQuery",                          //以属性方式存储信息
    value : "1.7.2"
};
jQuery[0] = "jQuery";                        //以数组方式存储信息
jQuery[1] = "1.7.2";
alert(jQuery.name);                          //返回 jQuery
alert(jQuery[0]);                            //返回 jQuery
</script>
```

上面的 jQuery 对象就是一个典型的 jQuery 对象, jQuery 对象的结构就是按这种形式设计的。可以说, jQuery 对象就是对象和数组的混合体, 但是它不拥有数组的方法, 因为它的数组结构是人为附加的, 也就是说它不是 `Array` 类型数据, 而是 `Object` 类型的数据。

- ☑ jQuery 对象包含的数据都是 DOM 元素, 通过数组形式存储, 即 `jQuery[n]` 形式获取。同时 jQuery 对象又定义几个模仿 `Array` 基本特性的属性, 如 `length` 等。

所以, jQuery 对象是不允许直接操作的, 只有分别读取它包含的每一个 DOM 元素, 才能够实现各种操作, 如插入、删除、嵌套、赋值、读写 DOM 元素属性等。

那么如何实现直接操作 jQuery 对象中的 DOM 元素呢? 在实际应用中, 可以看到类似下面的 jQuery 用法: `$("div").html()`

也就是说, 直接在 jQuery 对象上调用 `html()`, 并实现操作 jQuery 包含的所有 DOM 元素。那么这个功能是怎么实现的呢?

jQuery 定义了一个工具函数 `each()`, 利用这个工具可以遍历 jQuery 对象中所有的 DOM 元素, 并把需要操作的内容封装到一个回调函数中, 然后通过在每个 DOM 元素上调用这个回调函数即可。实现代码如下:

```
<div></div>
<div></div>
<div></div>
<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context) {
    return new jQuery.fn.init(selector, context);
}
jQuery.fn = jQuery.prototype = {
    init : function(selector, context) {           //省略的初始化构造器主体代码, 请参阅 18.2.5 节示例
    }                                              //定义 jQuery 对象方法
```

```

HTML: function(val){      //模仿 jQuery 框架中的 html()方法, 为匹配的每一个 DOM 元素插入 html 代码
    jQuery.each(this, function(val){      //调用 jQuery.each()工具函数, 为每一个 DOM 元素执行回调函数
        this.innerHTML = val;
    }, val);
}
}
jQuery.fn.init.prototype = jQuery.fn;
//扩展 jQuery 工具函数
jQuery.each = function( object, callback, args ){
    for(var i = 0; i<object.length; i++){
        callback.call(object[i],args);
    }
    return object;
}
//示例测试
$("div").html("测试代码");
</script>

```

在上面示例中, 通过为自己的 jQuery 对象绑定 html()方法, 然后利用 jQuery()选择器获取页面中所有的 div 元素, 接着调用 html()方法, 为所有匹配的元素插入 HTML 源码。

注意, 在上面代码中, each()函数的当前作用对象是 jQuery 对象, 故 this 指向当前 jQuery 对象, 即 this 表示一个集合对象。而在 html()方法中, 由于它是在指定 DOM 元素上执行的, 该函数内的 this 指针指向的是当前 DOM 元素对象, 即 this 表示一个元素。

当然, 上面示例所定义的 each()工具函数比较简陋, 适应能力比较有限。在 jQuery 框架中, 它封装的 each()函数功能就强大很多。具体代码如下:

```

jQuery.extend({
    //...
    //参数说明: object 表示 jQuery 对象, callback 表示回调函数, args 表示回调函数的参数数组
    each: function( object, callback, args ){
        var name, i = 0, length = object.length;
        if ( args ){
            //如果存在回调函数的参数数组
            if ( length === undefined ){
                //如果 object 不是 jQuery 对象
                for ( name in object ){
                    //遍历 object 的属性
                    if ( callback.apply( object[ name ], args ) === false )
                        //在对象上调用回调函数
                        break;
                    //如果回调函数返回值为 false, 则跳出循环
                }
            } else {
                //如果 object 是 jQuery 对象
                for ( ; i < length; )
                    //遍历 jQuery 对象数组
                    if ( callback.apply( object[ i++ ], args ) === false )
                        //在对象上调用回调函数
                        break;
                    //如果回调函数返回值为 false, 则跳出循环
            }
        } else {
            if ( length === undefined ){
                //如果 object 不是 jQuery 对象
                for ( name in object ){
                    //遍历 object 的属性
                    if ( callback.call( object[ name ], name, object[ name ] ) === false )
                        break;
                    //如果回调函数返回值为 false, 则跳出循环
                }
            } else {
                //如果 object 是 jQuery 对象
                for ( var value = object[0];
                    i < length && callback.call( value, i, value ) !== false; value = object[++i] ){
                    //遍历 jQuery 对象数组
                }
            }
        }
        return object;
    }
    //返回 jQuery 对象
}

```



```
//...
})
```

同时, jQuery 框架定义的 `html()` 方法如下所示。由于该方法包含的功能比较多, 它不仅可以插入 HTML 源代码, 还可以返回匹配元素包含的 HTML 源代码, 故它使用了一个条件结构分别进行处理。首先, 判断参数是否为空。如果为空则表示获取匹配元素中第 1 个元素包含的 HTML 代码, 则返回该 `innerHTML` 的值。如果不为空, 则先清空匹配元素中每个元素包含的内容, 并使用 `append()` 方法插入 HTML 源代码。

```
jQuery.fn = jQuery.prototype = {
  //...
  html: function( value ) {
    return value === undefined ?
      (this[0] ?
        this[0].innerHTML.replace(/ jQuery\d+="(?:\d+|null)"/g, "") :
        null) :
      this.empty().append( value );
  }
  //...
}
```

18.2.7 设计扩展接口

根据一般设计习惯, 如果为 jQuery 或者 `jQuery.prototype` 添加函数或方法时, 可以直接通过点语法, 或者在 `jQuery.prototype` 对象结构中增加一个属性即可。但是, 如果分析 jQuery 框架的源代码, 会发现它是通过 `extend()` 函数来实现功能扩展的。例如, 下面都是 jQuery 框架通过 `extend()` 函数扩展的功能:

```
jQuery.extend({                                //扩展工具函数
  noConflict: function( deep ) {},
  isFunction: function( obj ) {},
  isArray: function( obj ) {},
  isXMLDoc: function( elem ) {},
  globalEval: function( data ) {}
});
```

或者:

```
jQuery.fn.extend({                             //扩展 jQuery 对象方法
  show: function(speed,callback){},
  hide: function(speed,callback){},
  toggle: function( fn, fn2 ){},
  fadeTo: function(speed,to,callback){},
  animate: function( prop, speed, easing, callback ) {},
  stop: function(clearQueue, gotoEnd){}
});
```

这样做的好处是能够方便用户快速扩展 jQuery 框架的功能, 但是不会破坏 jQuery 框架的原型结构, 从而避免后期人工手动添加工具函数或者方法, 破坏了 jQuery 框架的单纯性, 同时也方便管理。如果不需要某个插件, 只需要简单地删除即可, 而不需要在 jQuery 框架源代码中去筛选和删除。

`extend()` 函数的功能实现起来也很简单, 它只是把指定对象的方法复制给 jQuery 对象或者 `jQuery.prototype`。例如, 在下面示例中, 为 jQuery 类和原型定义了一个扩展功能的函数 `extend()`, 该函数的功能很简单, 它能够把指定参数对象包含的所有属性复制给 jQuery 或者 `jQuery.prototype` 对象, 这样就可以在应用中随时调用它, 并动态扩展 jQuery 的方法。

```
<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context) {
```

```

    return new jQuery.fn.init(selector, context );
}
jQuery.fn = jQuery.prototype = {
    init : function(selector, context){

    }
}
jQuery.fn.init.prototype = jQuery.fn;
//jQuery 功能扩展函数
jQuery.extend = jQuery.fn.extend = function(obj) {
    for (var prop in obj) {
        this[prop] = obj[prop];
    }
    return this;
}
//扩展 jQuery 对象方法
jQuery.fn.extend({
    test : function(){
        alert("测试扩展功能");
    }
})
//测试代码
$("

").test();
</script>


```

在上面示例中，先定义一个功能扩展函数 `extend()`，然后为 `jQuery.fn` 原型对象调用 `extend()` 函数，为其添加一个测试方法 `test()`。这样就可以在实践中应用，如 `$("

").test()`。

jQuery 框架定义的 `extend()` 函数的功能要强大很多，它不仅能够完成基本的功能扩展，还可以实现对象合并等功能。详细代码和解释如下：

```

jQuery.extend = jQuery.fn.extend = function() {
    //定义复制操作的目标对象
    var target = arguments[0] || {}, i = 1, length = arguments.length, deep = false, options;
    //获取是否深度复制处理
    if ( typeof target === "boolean" ) {
        deep = target;
        target = arguments[1] || {};
        //跳出布尔值和目标对象
        i = 2;
    }
    //如果第 1 个参数是字符串，则设置为空对象
    if ( typeof target !== "object" && !jQuery.isFunction(target) )
        target = {};
    //如果只有一个参数，表示把参数对象的方法复制给当前对象，则设置 target 为 this
    if ( length == i ) {
        target = this;
        --i;
    }
    for ( ; i < length; i++ ) //遍历参数
        //当参数值不为 null，则进行处理
        if ( (options = arguments[ i ]) != null )
            //Extend the base object
            for ( var name in options ) { //遍历参数对象

```



```

var src = target[ name ], copy = options[ name ];
//防止死循环访问
if ( target === copy )
    continue;
//递归运算
if ( deep && copy && typeof copy === "object" && !copy.nodeType )
    target[ name ] = jQuery.extend( deep,
        //不要复制原对象
        src || ( copy.length != null ? [] : {} )
        , copy );
//不要传递未定义的值
else if ( copy !== undefined )
    target[ name ] = copy;
}
//返回修改后的对象
return target;
};

```

18.2.8 解决参数传递问题

在很多时候，jQuery 的方法都要求传递的参数为对象结构。例如：

```

$.ajax({
    type: "GET",
    url: "test.js",
    dataType: "script"
});

```

使用对象直接量作为参数进行传递，方便参数管理。当方法或者函数的参数长度不固定时，使用对象直接量作为参数存在很多优势。例如，对于下面的用法，ajax()函数就需要进行更加复杂的参数排查和过滤。

```
$.ajax("GET", "test.js", "script");
```

如果 ajax()函数的参数长度是固定的且是必需的，那么通过这种方式进行传递也就无所谓了。但是如果参数的个数和排序是动态的，那么使用\$.ajax("GET", "test.js", "script")这种方法是无法处理的。而 jQuery 框架很多方法都包含大量的参数，且都是可选的，位置也没有固定要求。所以使用对象直接量是唯一的解决方法。

使用对象直接量作为参数传递的载体，就涉及参数处理问题。解析并提出参数及处理参数和默认值可以通过下面方法来实现：

```

<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context) {
    return new jQuery.fn.init(selector, context);
}
jQuery.fn = jQuery.prototype = {
    init : function(selector, context) {},
    setOptions : function(options) {
        this.options = {                                //方法的默认值，可以扩展
            StartColor :    "#000",
            EndColor :      "#DDC",
            Background :    false,
            Step :          20,
            Speed :         10,
        };
    };
};

```

```

        jQuery.extend(this.options, options || {});           //如果传递参数，则覆盖原默认参数
    }
}
jQuery.fn.init.prototype = jQuery.fn;
jQuery.extend = jQuery.fn.extend = function(destination, source){    //重新定义 extend()函数
    for (var property in source) {
        destination[property] = source[property];
    }
    return destination;
}
</script>

```

在上面示例中，定义了一个原型方法 `setOptions()`，该方法能够对传递的参数对象进行处理，并覆盖默认值。这种用法在本书插件部分还将进行讲解，这里就不再展开。

在 jQuery 框架中，`extend()` 函数包含了所有功能，它既能够为当前对象扩展方法，也能够处理参数对象，并覆盖默认值。

18.2.9 设计名字空间

当一个页面中存在多个框架，或者自己写很多 JavaScript 代码，很难确保这些代码不发生冲突，因为任何人都无法确保自己非常熟悉 jQuery 框架中的每一行代码，难免会出现名字冲突，或者功能覆盖现象。为了解决这个问题，必须把 jQuery 封装在一个孤立的环境中，避免其他代码的干扰。

在详细讲解名字空间之前，先了解两个 JavaScript 概念。首先，请看下面代码：

```

var jQuery = function(){};
jQuery = function(){};

```

上面代码是两种不同的写法，且都是合法的，但是它们的语义完全不同。第 1 行代码声明了一个变量，而第 2 行代码是定义了 Window 对象的一个属性，也就是说它等同于下面语句：

```

window.jQuery = function(){};

```

在全局作用域中，变量和 Window 对象的属性可以是相等的，也是相通的。但是当在其他环境中，如局部作用域中，它们则是不相等的，也是无法互通的。因此如果希望 jQuery 具有类似下面的调用方式的能力：

```
$.method();
```

就需要将 jQuery 设置为 Window 对象的一个属性，所以就会看到 jQuery 框架中是这样定义的：

```

var jQuery = window.jQuery = window.$ = function( selector, context ){
    return new jQuery.fn.init( selector, context );
}

```

可能看到过下面的函数用法：

```

(function(){
    alert("观察我什么时候出现");
})();

```

上面就是一个典型的匿名函数基本形式。为什么要用到匿名函数呢？

这时就要进入正题了。如果希望自己的 jQuery 框架与其他任何代码完全隔离开来，也就是说把 jQuery 装在一个封闭空间中，不希望暴露内部信息，也不希望别的代码随意访问，匿名函数是一种最好的封闭方式。这样容易与其他代码冲突，此时只需要提供接口，方便与外界进行联系。例如，在下面示例中分别把 `f1` 函数放在一个匿名函数中，而把 `f2` 函数放在全局作用域中。最后，全局作用域中的 `f2` 函数可以允许访问，而匿名函数中的 `f1` 函数是禁止外界访问的。

```

<script language="javascript" type="text/javascript">
(function(){
    function f1(){

```

```

        return "f1()";
    };
})();
function f2(){
    return "f2()";
};
alert( f2() );           //返回 f2()
alert( f1() );           //抛出异常，禁止访问
</script>

```

实际上，上面的匿名函数就是所谓的闭包，闭包是 JavaScript 函数中一个最核心的概念。

当然，\$和 jQuery 名字并非是 jQuery 框架的专利，其他一些经典框架中也会用到\$名字，读者也可以定义自己的变量 jQuery。

在这之前需要让它与其他框架协同工作，因此带来一个问题，如果都使用\$作为简写形式就会发生冲突，为此 jQuery 提供了一个 noConflict()方法，该方法能够实现禁止 jQuery 框架使用这两个名字。为了实现这样的目的，jQuery 在框架的最前面，先使用_和_jQuery 临时变量寄存这两个变量的内容。当需要禁用 jQuery 框架的名字时，可以使用这个临时变量_和_jQuery 恢复\$和 jQuery 这两个变量的实际内容。实现代码如下：

```

(function(){
var
    window = this,
    undefined,
    _jQuery = window.jQuery,           //暂存 jQuery 变量内容
    _$ = window.$,                     //暂存$变量内容
    jQuery = window.jQuery = window.$ = function( selector, context ) {
        return new jQuery.fn.init( selector, context );
    },
    quickExpr = /^[^<]*(<(.|\s)+>)[^>]*$|^#(?:[w-]+)$/,
    isSimple = /^\.?:#(?:[w-]+)$/,
    jQuery.fn = jQuery.prototype = {
        init: function( selector, context ) {}
    }
})();

```

至此，jQuery 框架的设计模式就初见端倪了，后面的工作就是根据应用需要或者功能需要，使用 extend() 函数不断扩展 jQuery 框架的工具函数和 jQuery 对象的方法。

18.3 构建 jQuery 对象

前面两节从整体上分析了 jQuery 的设计思路和框架实现雏形，从其原理可以看出该框架统一入口就是 jQuery 对象。那么这个对象是如何生成的呢？

jQuery 实质是 Query，生成 jQuery 对象可以看作是构建并运行一个查询器。既然是查询，肯定会有查找结果（DOM 元素），那么这些结果又存放在哪里呢？

最好的地方当然是 jQuery 对象内。查询的结果可能是单个元素，也可能是集合，如 NodeSet。也就是说，jQuery 对象里面应该有一个集合，且这个集合是用来存放查询到的 DOM 元素的。但 jQuery 对象是所有操作的统一入口，那么它的构建就不应只局限于从 DOM 文档树中查询到 DOM 元素，有可能是从别的集合中转移过来的 DOM 元素，或是 HTML 片断生成的 DOM 元素。

jQuery 提供了 4 种构建方式，其中 jQuery 可以用\$代替：

- ☒ jQuery(expression,[context])

- ☑ jQuery(html)
- ☑ jQuery(elements)
- ☑ jQuery(callback)

这 4 种方式是经常用到的, 其实 jQuery 参数可以是任何元素, 也就是说任何参数都可以构建 jQuery 对象。例如:

```
$( "P" )
```

其参数可以是 jQuery 对象或 Array Like 集合。\$()是\$(document)的简写。\$(3)中, 把 3 放到 jQuery 对象集合中, 其中元素 (如 Array Like 集合元素) 不是 DOM 元素。jQuery 对象的方法设计的目的都是针对于 DOM 对象进行操作, 因此最好不要构建包含非 DOM 元素的 jQuery 对象。如果不清楚其使用, 很有可能会导致错误。下面从源码的角度分析 jQuery 对象。

通过 jQuery()构造函数直接调用, 实现没有生成对象, 它的 this 是指向 Window 对象的。那么 jQuery 的实例方法是怎样继承过来的呢? 看下面代码:

```
var jQuery = window.jQuery = window.$ = function(selector, context) {
    return new jQuery.fn.init(selector, context); ①
};
```

这是 jQuery 总入口, jQuery 对象不是通过 new jQuery()来继承其 prototype 中的方法, 而是 jQuery.fn.init 函数生成的对象。这里可以看出直接为 jQuery.prototype 添加函数集的对象的意义不大。new jQuery()还是可以的, 但是生成的 jQuery 对象在 return 时会被抛弃, 所以不要用 new jQuery()来构建 jQuery 对象。

jQuery 对象其实就是 jQuery.fn.init 对象, 那么 jQuery.fn.init.prototype 对象上就是挂着 jQuery 对象的操作方法。实现代码如下:

```
jQuery.fn.init.prototype = jQuery.fn;
```

那么 jQuery.fn.init.prototype 与 jQuery.fn.extend 的方法是什么关系呢? 这里是对 jQuery.fn 的引用, 在扩展 jQuery 时, 只要把相关函数扩展到 jQuery.fn 即可。

jQuery.fn.init 的工作方式如下:

```
//jQuery 原型对象
```

```
//构造 jQuery 对象的入口
```

```
//所有 jQuery 对象方法都通过 jQuery 原型对象来继承
```

```
jQuery.fn = jQuery.prototype = {
```

```
    //jQuery 对象初始化构造器, 相当于 jQuery 对象的类型, 由该函数负责创建 jQuery 对象
```

```
    //参数说明:
```

```
    //selector: 选择器的符号, 可以是任意数据类型, 考虑 DOM 元素操作需要, 该参数应该是包含 DOM 元素的任何数据
```

```
    //context: 上下文, 指定在文档 DOM 中哪个节点下开始进行查询, 默认值为 document
```

```
    init: function( selector, context ) {
```

```
        selector = selector || document;    //确保 selector 参数存在, 默认值为 document
```

```
        //第 1 种情况, 处理选择符为 DOM 元素, 此时将忽略上下文, 即忽略第 2 个参数
```

```
        //例如, $(document.getElementById("wrap")), jQuery (DOMElement)匹配 DOM 元素
```

```
        //先使用 selector.nodeType 判断当 selector 为元素节点, 将 length 设置为 1,
```

```
        并且赋值给 context, 实际上 context 作为 init 的第 2 个参数,
```

```
        也意味着它的上下文节点就是 selector 该点, 返回它的$(DOMElement)对象
```

```
        if (selector.nodeType) {    //存在.nodeType 属性, 说明选择符是一个 DOM 元素
```

```
            this[0] = selector;    //直接把当前参数的 DOM 元素存入类数组中
```

```
            this.length = 1;    //设置类数组的长度, 以方便遍历访问
```

```
            this.context = selector;    //设置上下文属性
```

```
            return this;    //返回 jQuery 对象, 即类数组对象
```

```
        }
```

```
        //如果选择符参数为字符串, 则进行处理
```

```
        //例如, $("<div>hello,world</div>"), jQuery(html,[ownerDocument])匹配 HTML 字符串
```



```

if (typeof selector == "string") {
    //使用 quickExpr 正则表达式匹配该选择符字符串，决定是处理 HTML 字符串，
    //还是处理 ID 字符串
    //quickExpr = /^[^<]*(<[^\s]+>)[^>]*$|^#([\w-]+)$/
    var match = quickExpr.exec(selector);
    //验证匹配的信息，任何情况下都不是 #id
    if (match && (match[1] || !context)) {
        //第 2 种情况，处理 HTML 字符串，类似 $(html) -> $(array)
        if (match[1])
            selector = jQuery.clean([match[1]], context);
        //第 3 种情况，处理 ID 字符串，类似 $("#id")
        else {
            var elem = document.getElementById(match[3]); //获取该元素
            //确保元素存在
            //处理在 IE 和 Opera 浏览器下根据 name 而不是 ID 返回元素
            if (elem && elem.id != match[3])
                return jQuery().find( selector ); //默认调用 document.find()方法
            //否则将把 elem 作为元素参数直接调用 jQuery()函数，返回 jQuery 对象
            var ret = jQuery( elem || [] );
            ret.context = document; //设置 jQuery 对象的上下文属性
            ret.selector = selector; //设置 jQuery 对象的上选择符属性
            return ret; //返回 jQuery 对象
        }
    } else
        //第 4 种情况，处理 jQuery(expression, [context])
        //例如，$("div.red")的表达式字符串
        return jQuery( context ).find( selector );
    } else if ( jQuery.isFunction( selector ) )
        //第 5 种情况，处理 jQuery(callback)，即$(document).ready()的简写
        //例如，$(function(){ alert("hello,world");})，或者$(document).ready(function(){ alert("hello,world");})
        return jQuery( document ).ready( selector );
    //确保旧的选择符能够通过
    if ( selector.selector && selector.context ) {
        this.selector = selector.selector;
        this.context = selector.context;
    }
    //第 6 种情况，处理类似$(elements)
    return this.setArray(jQuery.isArray( selector ) ?
        selector :
        jQuery.makeArray(selector));
}
//...
}

```

jQuery.fn.init 负责对传进来的参数进行分析然后生成 jQuery 对象。如果它的第 1 个参数是必需的（为空的，就是默认的 document）。从源码角度分析第 1 个参数有如下 4 种类型。

- ☑ DOM Element: 第 1 个参数为 DOM 元素，第 2 个参数不用。直接把 DOM 元素存在新生成的 jQuery 对象集合中。返回这个 jQuery 对象。构建 jQuery 对象完成。
- ☑ String: 第 1 个参数为字符串有 3 种情况。
 - html 的标签字符串，\$(html)->\$(array)，第 2 个参数可选。执行“selector = jQuery.clean([match[1]], context);”。该语句是把 html 的字符串转换成 DOM 对象的数组。接着执行 Array 类型的返回。

- 字符串为 #id 时, \$(id) 首先通过 “var elem = document.getElementById(match[3]);” 取得 elem。如没有取到, “selector = [];” 转到执行 Array 类型的返回空集合 jQuery 对象。如找到 elem, 通过 “return jQuery(elem);” 再次生成 jQuery 对象, 这次是 DOM Element 类型的 jQuery 对象的返回。
- 兼容 CSS 1~3 语法的 selector 字符串, 第 2 个参数是可选的。执行 “return jQuery(context).find(selector);”, 该语句先执行 jQuery(context)。可以看出 context 第 2 参数可以是任意的值, 可以是集合形式。之后再通过 find(selector) 找到 jQuery(context) 中所有 DOM 元素都满足 selector 表达式的 DOM 元素的集合, 构建新的 jQuery 对象并返回。#id 与这种方式是统一的, 单独列出来是为了提高性能。
- ☑ Fn: 第 1 个参数是函数。第 2 个参数不用。是 \$(document).ready(fn) 的简写, 其 return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector) 是其执行的代码。这个语句首先执行 jQuery(document), 它再一次调用 newjQuery.fn.init 函数, 生成 jQuery 对象 (元素为 document)。再调用这个对象的 ready(fn) 方法, ready(fn) 返回当前对象。而上面的语句又是返回这个 ready(fn) 的返回对象。可见这个 \$(fn) 返回的是 \$(document) 的对象。抛弃了第 1 次生成的 \$(fn) 对象。
- ☑ Array: 第 1 个参数是除上面提到的 DOM 元素外, 函数, string 以及所有其他的类型。可以为空, 如 \$()。第 2 个参数不用。语句 “return this.setArray(jQuery.makeArray(selector));” 首先把第 1 个参数转换为数组。Selector 可以是 Array-like 的集合, 如 jQuery 对象, 如 getElementsByTag 返回的 DOM 元素集合等, 可支持 \$(this)。Selector 还可能是单个任意的对象。转换成标准的数组之后, 执行 this.setArray 把这个数组中的元素全部存到当前 jQuery 对象的集合中。之后返回当前的 jQuery 对象。其实 DOM Element 完全可能综合在这里面, 单独拿出来是为了提高性能。

从上面的代码中可以看出, 构建 jQuery 对象就是往 jQuery 对象的集合中添加元素 (一般都应该是 DOM 元素)。添加的元素有两种形式:

- ☑ 单个元素, 可以通过直接的 DOM 元素的传参形式, 还可以通过 #id 从 DOM 文档中找元素。
- ☑ 集合, 如 jQuery 对象, 还有数组, 还有通过 CSS Selector 找到的 DOM 集合等 Array-Like。

18.4 构建 jQuery DOM 元素

在 jQuery.fn.init 函数中, 最终结果是把 DOM 元素存放到 jQuery 对象中的集合内。同时可以传入单个 DOM 元素或集合, 直接将其存入 jQuery 对象集合。如果第 1 个参数是字符串 (如 #id), 那么就到 DOM 文档树去查找。对于 HTML 片断的字符串, 将生成 DOM 元素。

那些传入 DOM 元素 (集) 的参数从哪里来呢? 它们可以通过 DOM 元素的直接或间接引用方式得到。本节首先分析如何从 HTML 片断生成 DOM 元素, 然后分析 jQuery 是如何通过直接或间接的方式在 DOM 树中找到 DOM 元素, 最后再分析基于 CSS 1~CSS 3 的 CSS selector。

18.4.1 生成 DOM 元素

jQuery.fn.init() 负责对传入参数进行分析, 然后生成并返回 jQuery 对象。jQuery.fn.init() 构造器的第 1 个参数是必需的, 如果为空, 则默认为 document。实际上使用 jQuery 选择器 (即 jQuery.fn.init() 构造器) 构建 jQuery 对象, 就是在 this 对象上附加 DOM 元素集合。附加的方式包括两类:

- ☑ 如果是单个 DOM 元素, 可直接把 DOM 元素作为数组元素传递给 this 对象, 还可以通过 ID 从 DOM 文档中查询元素。
- ☑ 如果是多个元素, 则以集合形式附加, 如 jQuery 对象、数组、对象等, 此时可以通过 CSS 选择器

匹配到所有 DOM 元素，然后经过过滤，最后构建类数组的数据结构。

而 CSS 选择器，则是通过 jQuery().find(selector) 函数来完成，通过 jQuery().find(selector) 可以分析选择器字符串，并在 DOM 文档树中查找到符合语法的元素集合（该函数将在下面章节进行分析），该函数能够兼容 CSS 1~CSS 3 选择器。

下面就从 init() 初始化构造器函数开始，来分析 jQuery 选择器是如何工作的。为方便解释，先结合源代码进行讲解：

```
//jQuery 原型对象
//构造 jQuery 对象的入口
//所有 jQuery 对象方法都通过 jQuery 原型对象来继承
jQuery.fn = jQuery.prototype = {
  //jQuery 对象初始化构造器，相当于 jQuery 对象的类型，由该函数负责创建 jQuery 对象
  //参数说明：
  //selector: 选择器的符号，可以是任意数据类型，考虑 DOM 元素操作需要，该参数应该是包含 DOM 元素的任何数据
  //context: 上下文，指定在文档 DOM 中哪个节点下开始进行查询，默认值为 document
  init: function( selector, context ) {
    selector = selector || document; //确保 selector 参数存在，默认值为 document
    //第 1 种情况，处理选择符为 DOM 元素，此时将忽略上下文，即忽略第 2 个参数
    //例如，$(document.getElementById("wrap"))，jQuery(DOMElement)匹配 DOM 元素。
    //先使用 selector.nodeType 判断当 selector 为元素节点，将 length 设置为 1，
    //并且赋值给 context，实际上 context 作为 init 的第 2 个参数，
    //也意味着它的上下文节点就是 selector 该点，返回它的$(DOMElement)对象
    if (selector.nodeType) {
      //存在 nodeType 属性，说明选择符是一个 DOM 元素
      this[0] = selector; //直接把当前参数的 DOM 元素存入类数组中
      this.length = 1; //设置类数组的长度，以方便遍历访问
      this.context = selector; //设置上下文属性
      return this; //返回 jQuery 对象，即类数组对象
    }
    //如果选择符参数为字符串，则进行处理
    //例如，$("#div>hello,world</div>")，jQuery(html,[ownerDocument])匹配 HTML 字符串
    if (typeof selector == "string") {
      //使用 quickExpr 正则表达式匹配该选择符字符串，决定是处理 HTML 字符串，还是处理 ID 字符串
      //quickExpr = /^[\^<]*(<(.|\s)+>)[^>]*$|^#([\w-]+)$/
      var match = quickExpr.exec(selector);
      //验证匹配的信息，任何情况下都不是 #id
      if (match && (match[1] || !context)) {
        //第 2 种情况，处理 HTML 字符串，类似$(html) -> $(array)
        if (match[1])
          selector = jQuery.clean([match[1]], context);
        //第 3 种情况，处理 ID 字符串，类似$("#id")
        else {
          var elem = document.getElementById(match[3]); //获取该元素
          //确保元素存在
          //处理在 IE 和 Opera 浏览器下根据 name 而不是 ID 返回元素
          if (elem && elem.id != match[3])
            return jQuery().find( selector ); //默认调用 document.find()方法
          //否则将把 elem 作为元素参数直接调用 jQuery()函数，返回 jQuery 对象
          var ret = jQuery( elem || [] );
          ret.context = document; //设置 jQuery 对象的上下文属性
          ret.selector = selector; //设置 jQuery 对象的上选择符属性
          return ret; //返回 jQuery 对象
        }
      }
    }
  }
};
```

```

    }
  } else
    //第 4 种情况, 处理 jQuery(expression, [context]),
    //例如, $("div.red")的表达式字符串
    return jQuery( context ).find( selector );
  } else if ( jQuery.isFunction( selector ) )
    //第 5 种情况, 处理 jQuery(callback), 即$(document).ready()的简写
    //例如, $(function(){ alert("hello,world");}), 或者$(document).ready(function(){ alert("hello,world");})
    return jQuery( document ).ready( selector );
  //确保旧的选择符能够通过
  if ( selector.selector && selector.context ) {
    this.selector = selector.selector;
    this.context = selector.context;
  }
  //第 6 种情况, 处理类似$(elements)
  return this.setArray(jQuery.isArray( selector ) ?
    selector :
    jQuery.makeArray(selector));
}
//...
}

```

进一步分析 init()构造器函数的设计思路:

(1) 如果第 1 个参数为 DOM 元素, 则第 2 个参数废除。直接把 DOM 元素存储到 jQuery 对象的集合中, 返回该 jQuery 对象。

(2) 如果第 1 个参数是字符串, 则可能存在以下 3 种情况。

- ☑ 第 1 个参数是 HTML 标签字符串, 第 2 个参数可选, 则执行 “selector = jQuery.clean([match[1]], context);”。该语句能够把 HTML 字符串转换成 DOM 对象的数组, 然后执行 Array 类型数组并返回 jQuery 对象。
- ☑ 第 1 个参数是 #id 字符串, 即类似 \$(id), 则先使用 document.getElementById()方法获取该元素。如果没有获得元素, 则设置 selector = [], 转到执行 Array 类型, 并返回空集合的 jQuery 对象。如果获得元素, 则构建 jQuery 对象并返回。这里, 把 #id 单独列出, 是为了提高性能。
- ☑ 处理复杂的 CSS 选择符字符串, 第 2 个参数是可选的。通过 “return jQuery().find(selector);” 语句实现, 该语句先执行 jQuery(context)。可以看出第 2 个参数 context 可以是任意值, 也可以是集合数据。然后调用 find(selector)找到 jQuery(context)上下文中所有的 DOM 元素, 即这些元素都满足 selector 表达式, 最后构建 jQuery 对象并返回。

(3) 如果第 1 个参数是函数, 则第 2 个参数可选。它是 \$(document).ready(fn)形式的简写, 其 return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector)是其执行的代码。该语句先执行 jQuery(document), 再通过 new jQuery.fn.init()方式创建 jQuery 对象, 此时元素为 document。再调用这个对象的 ready()方法, 并返回当前 jQuery 对象。\$(document).ready(fn)实现 DOMReady 的 jQuery 对象的统一入口, 可以通过 \$(fn)注册 domReady 的监听函数。所有的调用 jQuery 实现的功能代码都应该在 domReady 之后才能够运行。\$(fn)是所有的应用开发中的功能代码的入口, 它支持任意多的 \$(fn)注册。

(4) 如果第 1 个参数是除 DOM 元素、函数、字符串之外的所有其他类型, 也可以为空 (如 \$())。第 2 个参数可选, 则调用 “return this.setArray(jQuery.makeArray(selector));” 进行处理, 它先是把第 1 个参数转换为数组。当然这个参数可以是类数组结构的集合, 如 jQuery 对象, 如 getElementsByTagName 返回的 DOM 元素集合等, 可支持 \$(this)。selector 还可能是单个任意对象。转换成标准的数组之后, 执行 this.setArray 把这个数组中的元素全部存到当前 jQuery 对象集合中, 并返回 jQuery 对象。

18.4.2 间接引用 DOM 节点

jQuery.fn.init()构造函数能够构建 jQuery 对象，并把匹配的 DOM 元素存储在 jQuery 对象内部集合中。jQuery.fn.init()构造函数可以接收单个的 DOM 元素，也可以接收 DOM 集合。如果接收的是字符串型 ID 值，则将直接在文档中查找对应的 DOM 元素，并把它传递给 jQuery 对象。如果接收的是字符串型 HTML 片段，则需要把这个字符串片段生成 DOM 元素。下面将重点分析 jQuery 是如何把 HTML 片段生成 DOM 元素的。

前面提到 jQuery.fn.init()构造器通过“jQuery.clean([match[1]], context);”语句实现把 HTML 片段生成 DOM 元素，jQuery.clean()是一个公共函数。

jQuery.clean()包含 3 个参数，其中 elems 和 context 可以支持多种形式的值。elems 参数可以为数组、类数组、对象结构的形式。数组元素和对象属性可以混合使用。

对于数字类型，则会转换为字符串型，除了字符串型外，则都放入返回的数组中，当然对于集合形式只需要读取集合中每个元素即可。

对于字符串型参数，则把它转换成 DOM 元素，再存入返回的数组中。转换的方式是：把 HTML 字符串片段赋值给创建的 div 元素的 innerHTML，这样就可以把 HTML 字符串片段挂到 DOM 文档树中，从而实现把字符串转换成 DOM 元素。

在转换过程中，应该考虑 HTML 语法规约，因为标签嵌套是有严格限制的。例如，<td>必须存在于<tr>标签中，因此在执行转换前，还应该对 HTML 字符串进行预处理，即修正 HTML 标签不规范的用法，这也是 jQuery.clean()函数的一个重要工作。

//公共函数扩展

jQuery.extend({

 //把 HTML 字符串片段转换成 DOM 元素

 //参数说明：

 //elems 参数表示多个 HTML 字符串片段的数组

 //context 参数表示上下文

 //fragment 参数表示框架对象

 clean: function(elems, context, fragment) {

 context = context || document; //默认的上下文是 document

 //在 IE 中!context.createElement 是错误用法，因为它返回对象类型，而不是逻辑值，故通过返回类型进行判断

 if (typeof context.createElement === "undefined")

 //支持 context 为 jQuery 对象，并获取第 1 个元素

 context = context.ownerDocument || context[0] && context[0].ownerDocument || document;

 //如果只匹配一个标签，且没有指定框架参数，则直接创建 DOM 元素，并跳过后面的解析

 if (!fragment && elems.length === 1 && typeof elems[0] === "string") {

 var match = /^<(\w+)\s*V?>\$/i.exec(elems[0]);

 if (match)

 return [context.createElement(match[1])];

 }

 var ret = [], scripts = [], div = context.createElement("div");

 //匹配每一个 HTML 字符串片段，并为每一个片段执行回调函数

 jQuery.each(elems, function(i, elem){

 if (typeof elem === "number") //把数值转换为字符串的高效方法

 elem += " ";

 if (!elem)

 //如果不存在元素，则返回，或者为 undefined、false 等时也返回

 return;

 //把 HTML 字符串转换为 DOM 节点

 if (typeof elem === "string") {

 //统一转换为 XHTML 严谨型文档的标签形式，如<div/>的形式修改为<div></div>

```

//但(abbr|br|col|img|input|link|meta|param|hr|area|embed)不修改
//front=(<(\w+)[^>]*?)
elem = elem.replace(/(<(\w+)[^>]*?)\>/g, function(all, front, tag){
    return tag.match(/^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i) ?
        all :
        front + "</" + tag + ">";
});
//清除空格, 否则 indexOf 可能会不能正常工作
var tags = elem.replace(/\s+/, "").substring(0, 10).toLowerCase();
//有些标签必须是有一些约束的, 如<option>必须在<select></select>中间
//下面代码大部分用来对<table>中子元素进行修正。数组中第 1 个元素为深度
//&&>||
var wrap =
    //约束<option>, <opt 在开始位置上 (index=0)就返回&&运算符后面的数组
    !tags.indexOf("<opt") &&
    [ 1, "<select multiple='multiple'>", "</select>" ] ||
    //<leg 必须在<fieldset>内部
    !tags.indexOf("<leg") &&
    [ 1, "<fieldset>", "</fieldset>" ] ||
    //thead|tbody|tfoot|colg|cap 必须在<table>内部
    tags.match(/^(thead|tbody|tfoot|colg|cap)/) &&
    [ 1, "<table>", "</table>" ] ||
    !tags.indexOf("<tr") &&
    [ 2, "<table><tbody>", "</tbody></table>" ] ||
    //<tr 在<tbody>中间
    //td 在 tr 中间
    //col 在<colgroup>中间
    (!tags.indexOf("<td") || !tags.indexOf("<th")) &&
    [ 3, "<table><tbody><tr>", "</tr></tbody></table>" ] ||
    !tags.indexOf("<col") &&
    [ 2, "<table><tbody></tbody><colgroup>", "</colgroup></table>" ] ||
    //IE can't serialize <link> and <script> tags normally
    !jQuery.support.htmlSerialize &&
    //IE 中 link script 不能串行化
    [ 1, "div<div>", "</div>" ] ||
    //默认不修正
    [ 0, "", "" ];
//包裹 HTML 之后, 采用 innerHTML 转换成 DOM
div.innerHTML = wrap[1] + elem + wrap[2];
//转到正确的深度, 对于[1, "<table>","</table>"], div=<table>
while ( wrap[0]-- )
    div = div.lastChild;
//fragments 去掉 IE 对<table>自动插入的<tbody>
if ( !jQuery.support.tbody ) {
    //第 1 种情况: tags 以<table>开头但没有<tbody>。在 IE 中生成的元素中可能会自动加<tbody>
    //第 2 种情况: thead|tbody|tfoot|colg|cap 为 tags, 那 wrap[1] == "<table>"
    var hasBody = /<tbody/i.test(elem),
        tbody = !tags.indexOf("<table") && !hasBody ?
            div.firstChild && div.firstChild.childNodes :
            //tbody 不一定是 tbody, 也有可能是 thead 等
            wrap[1] == "<table>" && !hasBody ?
                div.childNodes :

```

```

        [];
        //除去<tbody>
        for ( var j = tbody.length - 1; j >= 0; --j )
            if ( jQuery.nodeName( tbody[j], "tbody" ) && !tbody[j].childNodes.length )
                tbody[j].parentNode.removeChild( tbody[j] );
        }
        //使用 innerHTML, IE 会去掉开头的空格节点, 因此加上去掉的空格节点
        if ( !jQuery.support.leadingWhitespace && /\s/.test( elem ) )
            div.insertBefore( context.createTextNode( elem.match(/^\s*/)[0] ), div.firstChild );
        //elem 从字符串转换成了数组
        elem = jQuery.makeArray( div.childNodes );
    }
    //如果是 DOM 元素, 则推入数组, 否则就合并数组
    if ( elem.nodeType )
        ret.push( elem );
    else
        ret = jQuery.merge( ret, elem );
});
//如果指定了第 3 个参数, 即框架对象, 则附加在框架对象上
//这段是新增加的用来处理 js 的代码, 同时也取消了 form 的处理
if ( fragment ) {
    for ( var i = 0; ret[i]; i++ ) {
        if ( jQuery.nodeName( ret[i], "script" ) && (!ret[i].type || ret[i].type.toLowerCase() ===
"text/javascript" ) ) {
            scripts.push( ret[i].parentNode ? ret[i].parentNode.removeChild( ret[i] ) : ret[i] );
        } else {
            if ( ret[i].nodeType === 1 )
                ret.splice.apply( ret, [ i + 1, 0 ].concat( jQuery.makeArray( ret[i].getElementsByTagName
("script") ) ) );
            fragment.appendChild( ret[i] );
        }
    }
    //返回脚本
    return scripts;
}
//返回 DOM 元素集合
return ret;
},
//...
}

```

上面这段代码实际上最后访问的是一个名为 `ret` 的数组, 数组中的元素变为 DOM 元素的对象, 而它的 `innerHTML` 正好就是刚才的 HTML 字符串。

18.4.3 采用 CSS 方式查找 DOM 节点

`jQuery()` 函数能够直接接收 HTML 字符串, 并把它们转换为 DOM 结构, 这是 18.4.2 节中讲解的 `jQuery()` 能够生成 DOM 元素。当然, 也可以看到 `jQuery()` 函数也可以接收 DOM 元素、DOM 元素集合、HTML 标签或者 ID 值。下面就来分析 `jQuery.fn.init()` 构造器是如何把这些类型参数转换为 DOM 元素的。

对于 HTML 标签来说, 使用 `document.getElementsByTagName()` 方法获取 DOM 元素集合。对于 ID 参数来说, 使用 `document.getElementById()` 方法获特定的 DOM 元素。另外, 还可以使用 DOM 元素的 `childNodes`、`firstChild`、`lastChild`、`nextSibling`、`parentNode`、`previousSibling` 等属性引用 DOM 节点。

既然说 DOM 元素能够通过 lastChild、parentNode 等属性引用节点，jQuery 对象又是 DOM 元素的集合。因此，jQuery 考虑通过整合 DOM 元素的这些属性来获得其集合中所有元素各自引用的节点。把这些间接引用的节点组合起来又构成了新的 DOM 元素集合。下面就来分析 jQuery 是如何引用节点的。代码如下：

//通过 each()方法为 jQuery 对象映射一组引用 DOM 节点的方法

```
jQuery.each({
    parent: function(elem){return elem.parentNode;},           //引用父亲节
    parents: function(elem){return jQuery.dir(elem,"parentNode");}, //引用所有父节点
    next: function(elem){return jQuery.nth(elem,2,"nextSibling");}, //引用相邻的下一个 DOM 元素
    prev: function(elem){return jQuery.nth(elem,2,"previousSibling");}, //引用相邻的上一个 DOM 元素
    nextAll: function(elem){return jQuery.dir(elem,"nextSibling");}, //引用所有后继 DOM 元素
    prevAll: function(elem){return jQuery.dir(elem,"previousSibling");}, //引用所有前继 DOM 元素
    siblings: function(elem){return jQuery.sibling(elem.parentNode.firstChild,elem);}, //引用相邻 DOM 元素
    children: function(elem){return jQuery.sibling(elem.firstChild);}, //引用所有子元素
    //如果存在 iframe，则就是文档，或者所有子节点
    //elem.contentDocument|| elem.contentWindow.document iframe 的属性
    //http://developer.mozilla.org/en/docs/XUL:iframe
    contents: function(elem){return jQuery.nodeName(elem,"iframe")?elem.contentDocument||elem.contentWindow.
document:jQuery.makeArray(elem.childNodes);}
}, function(name, fn){
    //为 jQuery 对象注册同名方法
    jQuery.fn[ name ] = function( selector ){
        //每个元素都执行同名方法
        var ret = jQuery.map( this, fn );
        //过滤元素集
        if ( selector && typeof selector == "string" )
            ret = jQuery.multiFilter( selector, ret );
        //返回构建的 jQuery 对象
        return this.pushStack( jQuery.unique( ret ), name, selector );
    };
});
```

上面代码中，为 jQuery 对象绑定了一组方法，它们能够引用不同位置的节点，主要包括父节点、子节点和兄弟节点 3 类操作方法：

- ☑ 对于父节点引用来说，可以获得当前 DOM 元素的父亲节点，也可以获得所有父级节点，即包括祖先节点。
- ☑ 对于子节点引用来说，就是所有直接的子节点，不包括不相邻的后代节点。
- ☑ 对于兄弟节点，包括当前 DOM 元素前或后的相邻节点，也包括前后相近的所有元素。

jQuery 通过 jQuery.each()公共函数把这 8 个节点引用的方法注册到 jQuery.fn 原型对象中，即 jQuery 对象中的同名方法中。因为 jQuery 对象的 DOM 元素是一个集合，所以还必须对集合中每个 DOM 元素执行相同的操作，也就是说为每个 DOM 元素调用属性包含的函数，如 parent: function(elem){return elem.parentNode;} 中的 function(elem){return elem.parentNode;}。

当然，在引用的节点中，还包括很多重复的 DOM 元素，或者用户需要过滤的其他节点，这些操作都需要利用过滤函数进行过滤。

在上面定义的 jQuery 对象方法中，jQuery 也提供几个公共函数：jQuery.dir()、jQuery.nth()和 jQuery.sibling()来辅助完成引用 DOM 节点。下面来分析这几个公共函数的用法。

//从一个元素出发，检索某个方向上的所有元素

//如可以把元素的 parentNode、nextSibling、previousSibling、lastChild、firstChild 属性作为方向

//参数说明：

//elem 参数表示起点元素


```
//dir 参数表示元素的方向属性，如 parentNode、nextSibling、previousSibling、lastChild、firstChild
jQuery.dir = function( elem, dir ){
    var matched = [], cur = elem[dir];
    //逐级迭代访问，直到 document 节点
    while ( cur && cur != document ) {
        if ( cur.nodeType == 1 )
            matched.push( cur );
        cur = cur[dir];    //向上一级传递节点
    }
    return matched;
};
```

dir 是 direction（方向）一词的缩写，表示朝一个方向一直迭代到尽头。例如，parentNode 能够把父节点作为当前节点，再取其父节点，通过这种方式可以迭代到 document 对象。另外，对于 nextSibling、previousSibling、lastChild 和 firstChild 元素属性都具有方向性。只要获取元素具有 dir（方向）特性的属性，就可以反复迭代读取。每循环一次都会把取到的元素保存起来。

所以说，dir()函数用于检索 DOM 文档树中呈放射线性排列的元素，是非常有用的。但是如果要检索在某个方向上的第几个元素，如检索偶数序号位置的元素，就需要使用 nth()函数。该函数的源代码如下：

```
//从一个元素出发，检索某个方向上的第几个元素。参数 Result 是第几个
//参数说明：
//cur 参数表示起点元素
//dir 参数表示元素的方向属性，如 parentNode、nextSibling、previousSibling、lastChild、firstChild
//result 参数表示级数，默认值为 1
//elem 参数是一个无用参数
jQuery.nth = function(cur, result, dir, elem){
    result = result || 1;
    var num = 0;
    for ( ; cur; cur = cur[dir] )
        if ( cur.nodeType == 1 && ++num == result )
            break;
    return cur;
};
```

jQuery.nth()函数与 jQuery.dir()函数在设计思路上是完全相同的，但是 jQuery.dir()函数不包含自身，而 jQuery.nth()函数可以包含自身。如果 jQuery.nth()函数的参数 result 等于 1，则返回自身元素。如果没有找到元素则返回空，如 undefined 或 null。

jQuery.sibling()函数则比较简单，没有上面两个方法实用。它实现从一个元素（包括自身）检索所有后续相邻元素，然后从这个后续的相邻元素排除一个指定的元素。代码如下：

```
//从包含参数 n 的元素开始的所有后续相邻元素，但不包含参数 elem 指定的元素
//参数说明：
//参数 n 表示起点元素
//参数 elem 排除元素
jQuery.sibling = function(n, elem){
    var r = [];
    for ( ; n; n = n.nextSibling ) {
        if ( n.nodeType == 1 && n != elem )
            r.push( n );
    }
    return r;
};
```

18.5 类 数 组

18.4 节分析了如何查找元素，查找后返回的结果可能是元素集合，也可能是单个对象，查到元素（集）之后 jQuery 就要将其保存，那么存在哪里？又该如何去存呢？存储后，又是如何读取或进行操作呢？本节将就这些话题进行讲解。

18.5.1 构建类数组

类数组（Array-Like）是 jQuery 框架核心概念，它描述了 jQuery 对象的基本形态。jQuery 选择器能够匹配一个或多个 DOM 元素，并把这些元素打包到一个数据集中返回，然后提供众多操作这个数据集的方法。类数组实际上就是对象，但是它类似数组。如果从集合角度分析，对象和数组其实都是样式，只不过一个是无序的，一个是有序的。而对于数组来说，它主要表现为带有元素下标和 length 属性。当数组元素增减时，length 属性会自动进行跟踪，反映这种变化。

jQuery 构造函数完成了查找、转换及其他功能，其结果就是查找到元素。不管是进行 DOM 树查找、将 HTML 字符串转换成 DOM 元素，还是直接传入 DOM 元素都只不过是方式而已。找到这些元素就得找个地方去存储起来。在 JavaScript 中最好的存储有序数据的地方（集合）当然是数组。在 jQuery 内实现数组可以采用如下的方式：

```
jQuery.fn.prototype=new Array();
在 this.setArray(arr)函数中添加：
Array.apply(this,arr);
如果再完美一点，还可以加上：
jQuery.fn.prototype.constructor=jQuery;
```

这样就既继承了数组的所有特性，又可以在 jQuery 对象中进行数组的功能扩展。但是 jQuery 并没有继承 Array 来实现内部集合，它采用了 Array-Like 对象来实现。

类数组的对象类似数组，但还是对象。数组与对象其实没有什么大的区别，有序和无序的集合是它们的区别。这个区别反应在数组包含 length 属性。当添加元素时，它会自动更新 length 属性值；当删除元素时，它会自动减去相对的个数。jQuery 按如下方式实现数组长度特性：

```
//第 1 种情况 Handle $(DOMElement)单个 DOM 元素，忽略上下文
if (selector.nodeType) {
    this[0] = selector;
    this.length = 1;
    return this;
}
```

这是它的第 1 种实现方法，通过 this[0]来直接设定第 1 个位置的 DOM 元素，同时设定 length=1。这里可以看出对象与数组一样都是采用 key/Value 对的形式存在对象中。上面的 JSON 形式为 {0: aDOM,length=1}。

这里详细分析一下数组。数组继承于对象，其[]的解释分析的最终结果可以看作是{}结构的对象，对[]或数组构建时会把 index(如 0,1,...)作为对象属性的 key。把数组中的值作为其对应的 value，同时改变 length 的值。这也就说明为什么本质上对象与数组没有多大的区别。在很多的源码中，如 YUI 中都采用对象的形式来构建多维数组。例如：

```
this.setArray(jQuery.makeArray(selector));
```

上面代码是第 2 种实现方法，第 1 种方法实现的是单个元素，这里实现多个元素的集合。它首先调用了 jQuery.makeArray(selector)静态方法把集合（类数组）转换成数组。

数组和对象都可以采用 obj.[attr]的形式来取得其 key 对应的 value。对于集合或类数组，必须要求其实

现 length 属性, 有了 length 的长度, 那么就从 0~length-1 的 key 属性中取得对应的 value 就可以了。代码如下:

```
//把类数组的集合转换成数组, 如果是单个元素就生成单个元素的数组
makeArray: function( array ) {
    var ret = [];
    if( array != null ){ var i = array.length;
        //单个元素, 但 window、string、function 有 length 的属性, 加其他的判断
        if( i == null || array.split || array.setInterval || array.call )
            ret[0] = array;
        else //类数组的集合
            while( i ) ret[--i] = array[i]; //Clone 数组
    }
    return ret;
},
```

生成了一个标准的数组, 那么接下来 setArray 做什么呢?

//把 array-like 对象的元素全部 push 到当前 jQuery 对象

```
setArray : function(elems) {
    this.length = 0; //初始化长度, 因为 push 会在原始的长度基础进行递增处理
    Array.prototype.push.apply(this, elems);
    return this;
},
```

调用 Array.prototype.push 来自动修改 length 的属性值 (当然是加入了元素)。由此可以想到, Array 中众多的方法 (如 shift) 都可以看作改变 length 的值, 在对象的 key/value 对中完成无序到有序或重新排序的工作。实际上 Array 等是采用 C 或 C++ 来实现的。但是它表现出 JavaScript 特性, 让我们可以思考采用 JavaScript 的实现方式。

setArray(elems) 函数只是会改变当前 jQuery 对象的集合, 它会清除这个对象集合中以前的元素。为了既保存原来集合中的元素, 同时也能对新传入的元素进行 jQuery 对象操作, 它提供了 pushStack() 函数来新建一个 jQuery 对象, 同时保存原来对象的引用, 这样就可能在需要时用到自己所要的对象。例如:

```
pushStack : function(elems) { //采用 jQuery 构建新对象, 同时引用老对象
    var ret = jQuery(elems); //构建新的 jQuery 对象
    ret.prevObject = this; //保存老的对象的引用
    return ret;
},
```

返回的是新建成的对象, 有着 jQuery 对象的全部功能, 同时还可以通过 prevObject 来访问原来的老对象。

18.5.2 操作类数组

类数组的操作主要包括元素定位、查找、复制和删除等。另外, 还可以通过迭代器和映射器扩展对类数组的操作功能。注意, 由于类数组的操作对象是集合, 因而与类数组包含的 DOM 元素操作是两个不同的概念。

1. 定位元素

jQuery 定义了 get() 和 index() 方法用来定位元素, 它们是集合操作的最基本方法。另外, jQuery 还定义了 get(index) 和 eq(index) 方法, 用于读取指定位置的元素。get(index) 方法和 eq(index) 方法的主要区别在于:

- ☒ get(index) 方法读取集合中的元素, 它与直接通过 [i] 来读取元素的方法是完全相同的。
- ☒ eq(index) 方法克隆集合中的元素, 也就是说不修改数组元素。

get() 方法的实现如下:

```
//获取 jQuery 对象的第几个 DOM 元素, 无参数时表示全部的 DOM 元素
get: function( num ) {
```

```

return num === undefined ?
//返回全部 DOM 元素的数组
Array.prototype.slice.call( this ):
//返回对应位置的 DOM 元素
this[ num ];
}

```

eq()方法的实现如下:

//获取 jQuery 对象的第几个 DOM 元素, 序号从 0 算起

```

eq: function(i) {
    //返回指定位置的元素
    return this.slice(i, +i + 1);
}

```

index()方法的实现如下:

//查找 elem 在 jQuery 对象的下标位置(index)

```

index: function( elem ) {
    return jQuery.inArray(
        //如果参数是 jQuery 对象, 则判断 jQuery 参数对象中第 1 个元素在当前 jQuery 对象中的位置
        elem && elem.jquery ? elem[0] : elem
        , this );
}

```

在 index()方法中, 调用 inArray()公共函数判断在当前类数组中 elem 的下标位置。inArray()函数的实现如下。index()方法支持的参数可以是 jQuery 对象或者 DOM 元素, 而 inArray()函数的参数可以是任何类型的元素。

```

//获取指定元素在数组中的下标位置
inArray: function( elem, array ) {
    for ( var i = 0, length = array.length; i < length; i++ )
        if ( array[ i ] === elem )
            return i;
    //如果不存在指定元素, 返回-1
    return -1;
}

```

2. 复制元素

jQuery 模拟 Array 的 slice()方法也实现元素复制功能。实现代码如下:

```

//模拟数组的 slice()方法
slice: function() {
    return this.pushStack( Array.prototype.slice.apply( this, arguments ),
        "slice", Array.prototype.slice.call(arguments).join(",") );
}

```

另外, 它还模拟数组的 concat()方法定义了一个全局函数 merge()。实现代码如下:

```

//模拟数组的 concat()方法
merge: function( first, second ) {
    //因为 IE 和 Opera 浏览器会重写 length 属性, 所以需要先存储 length 属性值
    var i = 0, elem, pos = first.length;
    //兼容 IE 浏览器
    if ( !jQuery.support.getAll ) {
        while ( ( elem = second[ i++ ] ) != null )
            if ( elem.nodeType != 8 )
                first[ pos++ ] = elem;
    } else

```



```

        while ( (elem = second[ i++ ]) != null )
            first[ pos++ ] = elem;
    return first;
},

```

3. 添加元素

jQuery 还为类数组定义了添加元素的方法。实现代码如下：

```

add: function( selector ) {
    return this.pushStack( jQuery.unique( jQuery.merge(
        this.get(),
        typeof selector === "string" ?
            jQuery( selector ) :
            jQuery.makeArray( selector )
        )));
}

```

上面方法能够把与表达式匹配的元素添加到 jQuery 对象中。类数组的集合也可以被追加进来。

4. 过滤元素

使用 add() 方法可以把其他元素增加到类数组中来，当然有时也需要过滤类数组中不需要的元素。jQuery 提供了 filter() 和 not() 方法来过滤元素。

filter() 方法能够筛选出与指定表达式匹配的元素集合，也可以通过该方法来筛选当前 jQuery 对象的元素，或者是使用逗号分隔的多个表达式。例如：

```

filter: function( selector ) {
    return this.pushStack(
        jQuery.isFunction( selector ) &&
        jQuery.grep( this, function( elem, i ) {
            return selector.call( elem, i );
        }) ||
        jQuery.multiFilter( selector, jQuery.grep( this, function( elem ) {
            return elem.nodeType === 1;
        }) ), "filter", selector );
}

```

filter() 方法是 grep() 和 multiFilter() 函数功能的综合。如果参数是函数，就采用 jQuery.grep() 函数来完成，否则采用 jQuery.multiFilter() 函数进行 selector 方式的过滤。

jQuery.grep() 函数提供了以自定义的函数回调的形式来过滤集合中不需要的元素，最后形成需要的数组，与 map() 函数功能类似。例如：

```

//过滤 elems 中满足 callback 处理的所有元素，inv 参数表示相反操作
grep: function( elems, callback, inv ) {
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ )
        if ( !inv != !callback( elems[i], i ) )
            ret.push( elems[i] );
    return ret;
},

```

jQuery.multiFilter() 函数与 jQuery.filter() 函数的区别不大。multiFilter 支持采用符号分隔的 selector 多表达式方式。例如：

```

jQuery.multiFilter = function( expr, elems, not ) {
    if ( not ) {
        expr = ":not(" + expr + ")";
    }
}

```

```
return Sizzle.matches(expr, elems);
};
```

jQuery.multiFilter()函数可以作为筛选器，与 jQuery.filter()函数一样，selector 的表达式也可以只是筛选器的组合，即以 *、#、:、[] 这 4 种符号做分隔的表达式。

not()方法也是根据 selector 来过滤不符合条件的元素，但是 not()方法是建立在 filter()方法基础之上的，执行效率会更高。实现代码如下：

```
not: function( selector ) {
    if ( isSimple.test( selector ) )
        return this.pushStack( jQuery.multiFilter( selector, this, true ), "not", selector );
    else
        selector = jQuery.multiFilter( selector, this );
    var isArrayLike = selector.length && selector[selector.length - 1] !== undefined && !selector.nodeType;
    return this.filter(function() {
        return isArrayLike ? jQuery.inArray( this, selector ) < 0 : this !== selector;
    });
}
```

5. 映射元素

集合映射是非常实用的工具。jQuery 定义了 each()和 map()两个映射方法，each()方法是对集合中每个元素都执行回调函数，而 map()方法还能够收集每个回调函数的返回结果组成一个新的集合。

对于 each()方法来说，jQuery 实现代码如下：

```
each: function( callback, args ) {
    return jQuery.each( this, callback, args );
}
```

然后通过调用 jQuery.each()公共函数实现元素的迭代操作：

//对 object 中的每个对象都执行 callback 函数

//参数 args 仅在内部使用

```
each: function( object, callback, args ) {
    var name, i = 0, length = object.length;
    if ( args ) {
        if ( length === undefined ) {
            for ( name in object )
                if ( callback.apply( object[ name ], args ) === false )
                    break;
        } else
            for ( ; i < length; )
                if ( callback.apply( object[ i++ ], args ) === false )
                    break;
        //不是类数组的 object，对每个属性进行 callback 函数的调用
    } else {
        if ( length === undefined ) {
            for ( name in object )
                if ( callback.call( object[ name ], name, object[ name ] ) === false )
                    break;
        } else
            for ( var value = object[0];
                i < length && callback.call( value, i, value ) !== false; value = object[++i] ) {}
    }
    return object;
}
```


这个公共函数支持第 1 个参数的类数组（数组）或对象，是数组就对每个元素进行 `callback` 操作。如果是对象，就对每个属性值进行 `callback` 操作。

`callback` 回调函数的语法格式如下：

```
callback:function(index,value)
```

参数 `index` 表示索引号，`value` 参数表示数组的 `index` 对应的元素或对象的第 `index` 个处理的属性。如果使用 `args` 参数，则 `callback` 回调函数的语法格式如下：

```
callback:function(args)
```

参数 `args` 是给回调函数设定的参数。而 `jQuery` 对象的 `each()` 方法的第 2 个参数 `args` 是采用传入的 `args` 直接给 `callback` 设定参数，而不是默认集合中 `index` 和对应的元素，但执行的次数还是集合的 `length` 次。

`map()` 方法能够将一组元素转换成其他数组，它是通过回调函数返回值组成的数组。实现代码如下：

```
map: function( callback ) {  
    return this.pushStack( jQuery.map(this, function(elem, i){  
        return callback.call( elem, i, elem );  
    }));  
},
```

该方法将一组元素转换成其他数组，然后根据这个数组构建新的 `jQuery` 对象。在该方法中，通过 `jQuery.map(this, function(elem, i){}` 语句中 `this` 指代的 `jQuery` 对象集合的每个元素当作回调函数的参数传到回调函数中。该回调函数又执行实例方法 `map()` 中的 `callback` 函数。`jQuery.map()` 通过代理的回调来取得转换而成的元素集合，然后采用 `pushStack()` 方法把该集合的元素构建成新的 `jQuery` 对象并返回，同时保存原 `jQuery` 对象的引用。

`map()` 公共函数的实现代码如下：

```
map: function( elems, callback ) {  
    var ret = [];  
    for ( var i = 0, length = elems.length; i < length; i++ ) {  
        var value = callback( elems[ i ], i );  
        if ( value != null )  
            ret[ ret.length ] = value;  
    }  
    return ret.concat.apply( [], ret );  
}
```

上面函数返回对 `elems` 每个元素都进行 `callback` 函数操作后所有返回值的集合。

18.6 Sizzle 引擎

Sizzle (<http://sizzlejs.com/>) 是 `jQuery` 的 DOM 选择器引擎。Sizzle 模拟 CSS 选择器设计思路。CSS 选择器 (CSS selector) 可以分为 3 种基本类型：ID 选择器 (`#id`)、Class 选择器 (`.class`) 和类型 (type) 选择器 (`p`)。另外，支持高级选择器，如属性 (attribute) 选择器、伪类或伪对象选择器 (Pseudo Classes) 等，这些都是单一的选择器，可以在应用中把它们组合起来，形成组合选择器，如 `div#id`, `div:last-child`。组合型选择器又包括多种形式，如包含关系、并列关系、相邻关系、父子关系等。

18.6.1 设计思路

为了方便讲解，下面结合一个选择器进行说明：

```
$("#div.red");
```

这是一个复合选择器，在 DOM 文档树中找到 `class` 属性等于 `red` 的 `div` 元素。根据选择器引擎的工作方

式，分两步来实现：

(1) 根据 `document.getElementsByTagName()` 方法选择文档树中的 `div` 元素集合。

(2) 根据其 `class` 是否等于 `red` 进行判断，把不等于该值的元素从结果集中去掉。

不仅是这个选择器，对于所有形式的复合选择器，都可以根据这种思路来拆分选择器，然后分别完成每一部分的操作。jQuery 选择器引擎不同版本在设计思路上也存在一些细微的区别。例如，针对下面的选择器：

```
$("#div p");
```

早期的 jQuery 选择器是根据下面步骤进行解析的：

(1) 根据 `document.getElementsByTagName()` 方法选择文档树中的 `div` 元素集合。

(2) 迭代 `div` 元素集合，在所有的 `div` 元素中查找每个 `div` 元素下的 `p` 元素。

(3) 合并结果。

Sizzle 选择器调整了解析顺序：

(1) 根据 `document.getElementsByTagName()` 方法选择文档树中的 `p` 元素集合。

(2) 迭代 `p` 元素集合，在所有的 `p` 元素中查找每个 `p` 元素的父级元素。

(3) 检测父级元素，如果不是 `div` 元素，则遍历上一级元素。如果迭代到文档树的顶层，则排除该 `p` 元素。如果是 `div` 元素，则保存该 `p` 元素。

Sizzle 选择器放弃了合并结果操作，直接在遍历过程中过滤元素，所以导致查找速度大幅提升。那么 jQuery 在匹配元素时如何工作呢？

例如，对于 `$("#div[id=value]")` 选择器，JavaScript 先匹配 `div` 元素，然后判断 `div` 元素的 `id` 属性是否等于 `value`，如果不等于就从结果集中删除掉。而对于 `$("#div#value]")` 选择器也是一样。根据上面的分析，CSS 选择器可以分为两部分。

☑ 选择器 (selector)：根据给定的选择符，从 DOM 文档树找到相关的元素节点，并存储到结果集中。

☑ 过滤器 (filter)：根据表达式的条件，在结果集中过滤元素。

对于类型选择器来说，由于可以直接使用 `document.getElementsByTagName()` 方法进行选择，因此，直接使用选择即可。类型选择器相互之间还可以组成各种形式的复合选择器，如 `*`、`E F`、`E~F`、`E+F`、`E>F`、`E/F`、`E`。

虽然这些特殊形式的类型选择器由多个选择器构成，但是它们都可以从文档中直接选择，可以统一把它们归为选择器类型。而对于 ID 选择器和 Class 选择器来说：

☑ 如果它们在 `selector` 字符串的起始位置，那么也可以完成选择(selector)功能，如 `$("#id")`、`$(".calss")`。

☑ 如果它们不在起始位置，那么就应该作为筛选器实现，如 `$("#div#id")`、`$("#div.calss")`。

实际上，对于 ID 选择器可以直接调用 JavaScript 的 `document.getElementById()` 方法进行选择。但是对于 `.class` 选择器来说，由于它无法直接进行选择。因此，可以把 ID 选择器视为选择器，而 Class 选择器视为过滤器。

对于 Class 选择器来说，还可以把它转换为 `*.class` 形式。其中，`*` 表示选取范围内的所有元素，然后再进行过滤。这样就可以实现统一的编程接口。

对于属性选择器、伪类选择器来说，它们只能附在其他选择器后面使用。如果单独使用，则可以通过在前面添加 `*` 标签，以便设计成统一的语法格式，以方便选择器引擎的工作。这与 Class 选择器的工作方式是相同的，即都视为过滤器。

例如，对于下面这个复杂的选择器来说，包含了类型选择器、Class 选择器、ID 选择器、属性选择器和伪类选择器。

```
$("#div.red:nth-child(odd)[title=bar]#wrap p");
```

jQuery 解析的步骤如下：

(1) 选择 DOM 文档树中所有的 `p` 元素，建立初步结果集。

- (2) 在结果集中, 选择父级元素为 `div` 的元素, 形成新的结果集。
- (3) 在新的结果集中筛选 `class` 属性为 `red` 的元素。
- (4) 解析伪类选择器 `:nth-child(odd)`, 在结果集中筛选元素的子元素为偶数的元素。
- (5) 解析属性选择器 `[title=bar]`, 在结果集中筛选元素的 `title` 属性为 `bar` 的元素。
- (6) 在结果集中筛选 `id` 等于 `wrap` 的元素。

18.6.2 设计框架

Sizzle 引擎在 jQuery 框架中处于核心地位, 如图 18.1 所示。在下面的 jQuery 选择器逻辑流程图中, 首先, 对传入的选择符参数进行过滤, 只有为表达式字符串时, 才会进入 `jQuery.fn.find()` 入口, 然后进入 Sizzle 接口 (`jQuery.find()`), 在 Sizzle 构造器中分别调用 `Sizzle.find()` 和 `Sizzle.filter()` 函数完成选择和过滤操作。

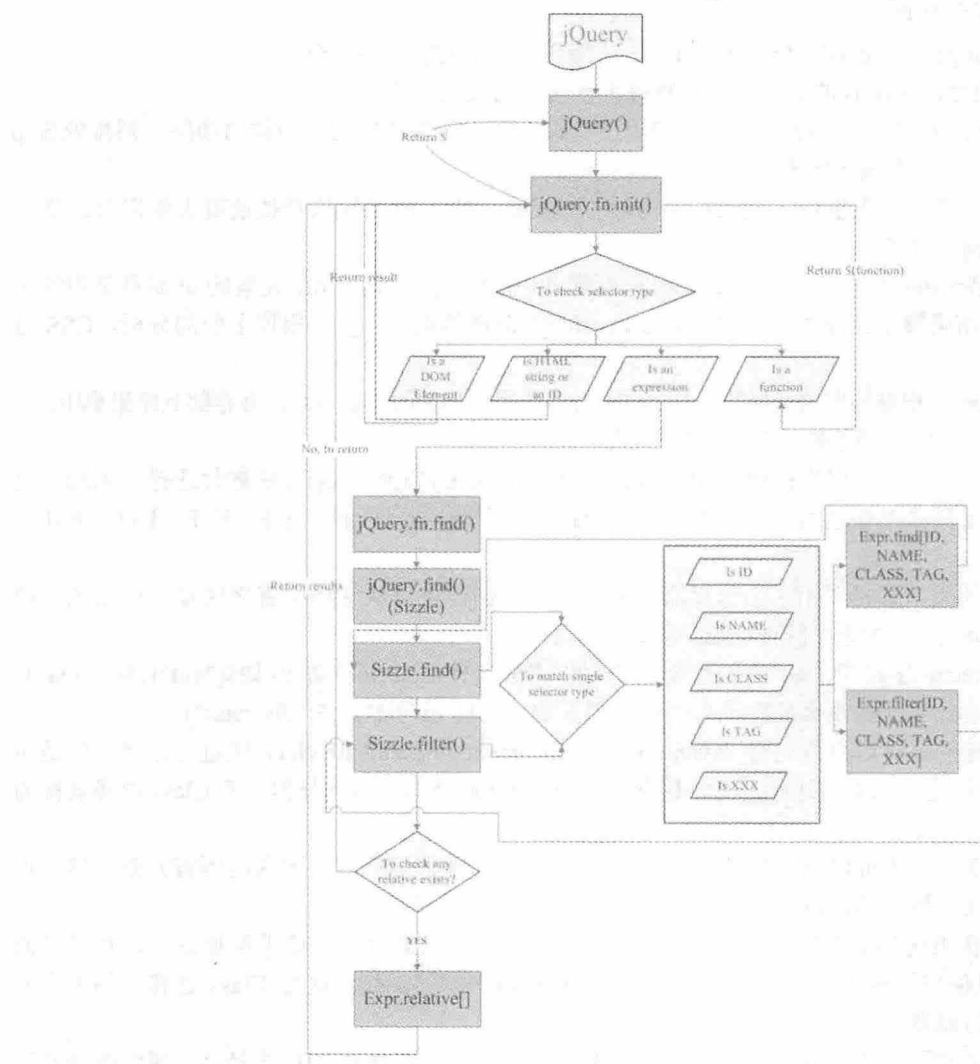


图 18.1 Sizzle 引擎工作流程图

在选择 (`Sizzle.find()`) 和过滤 (`Sizzle.filter()`) 操作过程中, 都会经过这样的处理流程: (To match single selector type, 匹配简单的选择器类型) ID 选择器、Name、Class 选择器、类型选择器……

对于选择器 (`Sizzle.find()`) 来说, 它会调用 `Expr.find[type]`。而对于过滤器 (`Sizzle.filter()`) 来说, 它

会调用 `Expr.filter[type]`，最后分别返回 `Sizzle.find()` 和 `Sizzle.filter()`。

在返回 `Sizzle.filter()` 的过程中，它还需要检测关系选择符是否存在（To check any relative exists）。如果存在，则调用 `Expr.relative[]`，最后返回结果集。

下面对 Sizzle 引擎的主体结构进行简单的分析：

```

/*!
 * Sizzle CSS Selector Engine - v0.9.3
 * Copyright 2009, The Dojo Foundation
 * Released under the MIT, BSD, and GPL Licenses.
 * More information: http://sizzlejs.com/
 */
//把 Sizzle 引擎封装在一个独立的空间中
(function(){
//定义用于块识别器的正则表达式
var chunker = /((?:\((?!\([^\)]*\)|["'](?:\\.|[^\s()"]*)\)|\/(?:\(|\/|[^\s()\/]|\\.)\)|[^\s()\/|"]+)|\s+|>+~|(\s*,\s*\s*))?(?!\s|\)|\/|>+~|(\s*,\s*\s*))?/g,
    done = 0,
    toString = Object.prototype.toString;
//Sizzle 选择器引擎构造器函数
//参数说明：
//selector: 选择器字符串
//context: 上下文
//results: 结果集
//seed: 种子
var Sizzle = function(selector, context, results, seed) {
    //省略的函数体
};
//Sizzle 匹配函数
//参数说明：
//expr: 匹配表达式
//set: 条件设置选项
Sizzle.matches = function(expr, set){
    return Sizzle(expr, null, null, set);
};
//Sizzle 查询函数
//参数说明：
//expr: 查询表达式
//context: 上下文
//isXML: 检测函数
Sizzle.find = function(expr, context, isXML){
    //省略的函数体
};
//Sizzle 过滤函数
//参数说明：
//expr: 过滤表达式
//set: 条件设置选项
//inplace: 包含项
//not: 排除项
Sizzle.filter = function(expr, set, inplace, not){
    //省略的函数体
};
//Sizzle 表达式对象
//列举所用的各种匹配表达式

```

* /

```

//Sizzle 选择器引擎构造器函数
//参数说明:
//selector: 选择器字符串
//context: 上下文
//results: 结果集
//seed: 种子
var Sizzle = function(selector, context, results, seed) {
    results = results || [];           //设置默认结果集为空
    context = context || document;     //设置上下文为 document 对象
    //如果上下文不是元素和文档对象, 则返回空集合
    if ( context.nodeType !== 1 && context.nodeType !== 9 )
        return [];
    //如果选择器参数不存在, 或者不为字符串类型, 则返回默认结果集
    if ( !selector || typeof selector !== "string" ) {
        return results;
    }
    //初始化变量
    var parts = [], m, set, checkSet, check, mode, extra, prune = true;
    //重置 chunker 正则表达式匹配的起始位置为开始位置
    chunker.lastIndex = 0;
    //匹配选择符字符串, 如果存在则执行下面操作
    while ( (m = chunker.exec(selector)) !== null ) {
        parts.push( m[1] );           //存储匹配的字符串信息
        //如果是选择符组, 则获取后半部分
        if ( m[2] ) {
            extra = RegExp.rightContext;
            break;
        }
    }
    //Expr 和 Sizzle.selector 指向同一个对象, 这个选择器对象与老版本不一样
    //放弃了 if else 语句判断, 全部使用正则表达式进行处理, 执行效率提高了很多
    //具体代码参阅 Expr 对象
    //POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*\))?)?(?=[^~]|$)/,
    //下面代码判断选择符字符串中是否存在特殊函数
    if ( parts.length > 1 && origPOS.exec( selector ) ) {
        //relative 是函数对象, 封装了+>~函数, 表示 typeof Expr.relative[ parts[0] ] ==Function
        if ( parts.length === 2 && Expr.relative[ parts[0] ] ) {
            set = posProcess( parts[0] + parts[1], context );
        } else {
            set = Expr.relative[ parts[0] ] ?
                [ context ] :
                Sizzle( parts.shift(), context );
            //如果匹配到>+~, 过滤上下文, 去掉选择器中的符号, 再分析选择器
            while ( parts.length ) {
                selector = parts.shift();
                if ( Expr.relative[ selector ] )
                    selector += parts.shift();
                set = posProcess( selector, set );
            }
        }
    }
    } else {
        //执行种子操作

```



```

var ret = seed ?
    { expr: parts.pop(), set: makeArray(seed) } :
    Sizzle.find( parts.pop(), parts.length === 1 && context.parentNode ? context.parentNode : context,
isXML(context) );
set = Sizzle.filter( ret.expr, ret.set );
if ( parts.length > 0 ) {
    checkSet = makeArray(set);
} else {
    prune = false;
}
while ( parts.length ) {
    var cur = parts.pop(), pop = cur;
    if ( !Expr.relative[ cur ] ) {
        cur = "";
    } else {
        pop = parts.pop();
    }
    if ( pop == null ) {
        pop = context;
    }
    Expr.relative[ cur ]( checkSet, pop, isXML(context) );
}
}
//设置默认值
if ( !checkSet ) {
    checkSet = set;
}
//抛出异常
if ( !checkSet ) {
    throw "Syntax error, unrecognized expression: " + (cur || selector);
}
//如果检测选项 checkSet 是数组，则执行下面操作
if ( toString.call(checkSet) === "[object Array]" ) {
    //如果 checkSet 没有包含 set 选项值
    if ( !prune ) {
        //则把 checkSet 推入到结果集中
        results.push.apply( results, checkSet );
    } else if ( context.nodeType === 1 ) { //如果上下文是元素类型的对象
        for ( var i = 0; checkSet[i] != null; i++ ) { //则遍历检测选项集，然后把 set 推入结果集中
            if ( checkSet[i] && (checkSet[i] === true || checkSet[i].nodeType === 1 && contains(context,
checkSet[i])) ) {
                results.push( set[i] );
            }
        }
    }
} else {
    //则遍历检测选项集，然后把 set 推入结果集中
    for ( var i = 0; checkSet[i] != null; i++ ) {
        if ( checkSet[i] && checkSet[i].nodeType === 1 ) {
            results.push( set[i] );
        }
    }
}
}

```

```

    } else {
        //把结果集与检测选项组合并为数组
        makeArray( checkSet, results );
    }
    //处理组选择器中下一个选择器
    if ( extra ) {
        //把后半部分选择器字符串再次传递给构造器，执行下一次匹配处理
        Sizzle( extra, context, results, seed );
        //如果存在排序函数，则调用它对结果集进行排序
        if ( sortOrder ) {
            hasDuplicate = false;           //指示已经排序
            results.sort(sortOrder);         //排序结果集
            //如果没有排序，则对结果集进行重排
            if ( hasDuplicate ) {
                for ( var i = 1; i < results.length; i++ ) {
                    if ( results[i] === results[i-1] ) {
                        results.splice(i--, 1);
                    }
                }
            }
        }
    }
    //返回匹配的结果集
    return results;
};

```

Sizzle 过滤器主要包含两部分：第 1 部分是过滤函数 (jQuery.filter())，在该函数中将对需要过滤的表达式及其对应的表达式处理函数执行分析，并返回过滤后的 jQuery 对象；第 2 部分就是过滤表达式对象 (Expr = Sizzle.selectors)，该对象包含了所有表达式处理的方法和匹配的正则表达式。

Sizzle 选择器先初步分析参数字符串，并找出基本特征字符串，并根据这些特征字符在 Expr 对象中调用对应的正则表达式，然后进一步分析选择符字符串，同时根据进一步分解出来的特殊字符，在 Expr 对象中匹配到对应的选择器函数，最后根据这个选择器函数的返回值是否为 true，决定是否保留当前过滤的元素，被保留的元素就是筛选元素，它被推进了 jQuery 对象集合中。

jQuery.filter() 函数完成分析属性 ([])、Pseudo (:)、Class (.) 和 ID (#) 的筛选功能，从给定的集合中筛选出满足上面 4 种筛选表达式的集合。针对 find() 方法，jQuery.filter() 函数的完成并不表明整个选择符的分析完成了，还会交替地通过查找器来查找或通过该函数来筛选。对于单独使用这个函数，表达式中就不应该含有查找的选择器表达式了。筛选是根据 []、:、#、. 这 4 个符号来作为筛选器的分隔符的。Class 筛选器是通过 classFilter 来完成的。它还把 Pseudo 中的 :not、:nth-child 从 Pseudo 类中单独提出来处理。对于 [的属性筛选器，实现起来也很简单。除了这些，它还调用 jQuery.expr[m[1]] 来处理 Pseudo 类。



清华社“视频大讲堂”大系

网络开发视频大讲堂

执着于专业，精细于品质



ISBN 978-7-302-30667-2



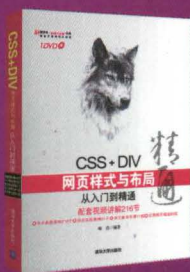
定价：79.80元



ISBN 978-7-302-30881-2



定价：69.80元



ISBN 978-7-302-30671-9



定价：69.80元



ISBN 978-7-302-30666-5

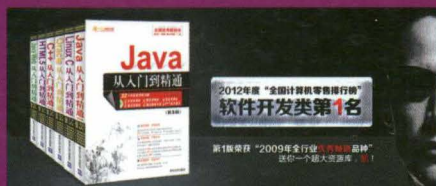


定价：79.80元

清华社“视频大讲堂”大系简介

“视频大讲堂”大系是清华社第六事业部重点打造的精品大系，该大系自2008年出版“软件开发视频大讲堂”以来，先后出版了多个子系列，本着“宁缺勿滥，件件精品”的原则，该大系取得了如下业绩：

- 4个品种荣获“全行业优秀畅销品种”
- 1个品种荣获2012年清华大学出版社“专业畅销书”一等奖
- 绝大多数品种在“全国计算机零售图书排行榜”同品种排行中名列前茅
- 截至目前该大系累计销售超过55万册
- 该大系已成为近年来清华社计算机专业基础类零售图书最畅销的品牌之一



清华大学出版社数字出版网站

WQBook 书文局景

www.wqbook.com

ISBN 978-7-302-30666-5



9 787302 306665 >

定价：79.80元

(1DVD, 含配套视频、参考手册、网页模板、素材源程序等)